Detecting Non-Termination in Constraint Handling Rules

DETECTING NON-TERMINATION IN CONSTRAINT HANDLING RULES

By
ERSHAD RAHIMIKIA, B.SOFTWARE ENGINEERING

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements for the Degree of

Master of Science

Department of Computing and Software
McMaster University

© Copyright by Ershad Rahimikia, September 24, 2007

MASTER OF SCIENCE(2007)
(Computing and Software)

McMaster University
Hamilton, Ontario

TITLE: Detecting Non-Termination in Constraint Handling Rules

AUTHOR: Ershad Rahimikia, B.Software Engineering
(University of Tehran)

SUPERVISOR: Dr. Wolfram Kahl

NUMBER OF PAGES: iv, 61

Abstract

Constraint Handling Rules (CHRs) are a high level language extension to introduce user-defined constraints into a host language. Application of CHRs to reformulate functional dependencies (FDs) in the Haskell type system gives us a more precise definition of this concept, and a better understanding of FD behavior. But to preserve the confluence and termination properties of CHRs generated from FDs, some restrictions on the syntax of FDs and type class definitions have been imposed which confines the expressiveness power of Haskell type system.

In this thesis we use this problem as a motivation to find a solution for the confluence and non-termination problem in CHRs. We build a formal framework for CHRs and model their different aspects mathematically to study how non-confluence and non-termination happens. Based on this formalization we introduce prioritized CHRs as a solution for the confluence problem. To solve the non-termination problem, we propose a method to detect non-termination in the constraint solver. We define a *repetition candidate* as a special type of derivation and prove that a derivation having this property can cause non-terminating rule applications in the system. Finally we define a deduction tree structure for a set of rules that can be used to find all the possible repetition candidates for a set of constraint rules.

Acknowledgements

I am grateful to my supervisor Dr. Wolfram Kahl, the best math teacher I have ever had, for his helpful inspirations and guidance and his patience in careful reviewing of my work.

I also wish to thank my parents for their unconditional encouragement and support throughout my life.

If you're not failing every now and again, it's a sign you're not doing anything very innovative.

Woody Allen

Contents

1 Introduction 3

2 Background 8

2.1 Type Classes in Haskell 8

2.2 Implementing Type Classes in Haskell 9

2.3 Multi-parameter Type Classes and Functional Dependencies . 11

2.4 CHR and Functional Dependencies 12

2.4.1 Constraint Handling Rules 13

2.4.2 CHR as a Term Rewriting System 15

2.4.3 Formalizing Functional Dependencies with CHRs . . . 16

2.5 An Alternative Approach: Associated Type Synonyms 20

2.5.1 Associated Types 20

2.5.2 Associated Type Synonyms and Type Dependencies . . 22

2.5.3 Comparing the Two Approaches 23

3 A Formal Framework for CHRs 25

3.1 Substitution, Unification, Matching 26

3.2 Free Variable Instantiation 28

3.3 Primitive Constraints 29

3.4 Propagation Rule Applications and Derivations 32

4 The Confluence Problem 35

4.1 Problem Description 35

4.2 Prioritized CHRs 36

4.3 Prioritized CHRs by Examples 37

4.3.1 Example 1 Coverage Condition 38

4.3.2 Example 2 Consistency Condition 39
4.3.3 Example 3 40
4.4 Prioritized CHR and Confluence 41

5 The Non-Termination Problem 43
5.1 Matching Constraints and Infinite Derivations 43
5.2 Finding Repetition Candidates 52
5.2.1 Deduction Trees Finiteness 55
5.2.2 Building the Deduction Tree 56
5.2.3 Example 56

6 Conclusions 58
6.1 Contributions 59
6.2 Future Work 60

Chapter 1

Introduction

Functional dependencies (FDs) are a useful extension to Haskell type classes that add to the expressiveness of the language and resolve ambiguities in type definitions. In relational algebra terminology, when a functional dependency exists between two attributes in a relation one attribute uniquely determines the other attribute. Mapping the same concept to type classes, having a functional dependency between two type parameters in a type class, one type parameter uniquely determines the other.

Based on this informal definition, it is possible to implement functional dependencies as part of the type system, but formalizing functional dependencies in a more precise way enables us to study their properties and better understand their behavior. One candidate to do that is a high level language extension called constraint handling rules (CHRs). This formalization can be useful for practical reasons too; by translating functional dependencies to CHRs, part of the type inference can actually be done by the constraint solver.

Any CHR system basically consists of a set of rule definitions that shows how the existence of a set of constraints entails other constraints in the system. Starting from an initial constraint set, the constraint solver applies these rules and finds all the possible constraints that can be inferred or detects any inherent inconsistencies in the initial set.

Formalization of FDs is based on a set of translation rules that shows how every type class or instance definition can be translated to CHRs. But

to have a sound, complete and decidable type inference system that supports functional dependencies, we need to ensure that the generated CHRs are terminating and confluent. These two properties in fact come from the more general context of term rewriting systems, and a great deal of research has been done to find how it can be guaranteed that a term rewriting system is terminating and confluent.

The current approach to ensure termination and confluence of the generated CHRs from type class and instance definitions is to put some constraints on the definition of functional dependencies. This has been done by defining three basic properties that all type class and instance definitions should satisfy, namely consistency, coverage and bound variable conditions.

Although these limitations make the resulting CHRs system have the the basic required properties, in many cases they are too limiting and reject many useful and correct programs. To solve or at least to mitigate this problem, some less constraining conditions, such as weak coverage conditions, have been found to make the system encompass more programs as valid. But these weaker conditions are more complex than the original ones and usually need some supplementary conditions to assure the basic CHRs properties are satisfied. The main problems that we see in the current approach are listed below:

- The limitations of using CHRs to implement functional dependencies are excessively affecting decisions for the language syntax and semantics.

- The complex rules to define a type class or an instance can result in a sound, complete and decidable inference system, but these limitations have a direct effect on the language syntax and are not hidden from the programmer. So the programmer has to take care of many rules when defining classes and instances, many of which have no intuitive rationale.

- There is no guarantee that even with the new looser conditions we will not have reasonable definitions rejected by the system, and this inevitably leads to a cycle of adding more refined but usually more complex conditions.

- The current preemptive approach is too conservative in that a class, instance or function definition that "might" be used in a wrong way is rejected. Giving the error when they are actually used in a wrong way is another approach that should be considered.
- CHRs are not fully integrated with the rest of the typing system. Supporting functional dependencies as a part of the type inference system is another approach that can be worked on.

To avoid these problems, some research has recently been done to find an alternative way to formalize functional dependencies. As we will see in the next chapter, *associated type synonyms* can be used to formalize functional dependencies as an integral part of the typing system [CKPJ05]. Using this new method enables us to include more class and instance definitions as valid, but still it does not accept all the reasonable definitions. Furthermore, supporting associated type synonyms in typing systems considerably adds to the complexity of the inference rules.

Another approach to deal with the problems mentioned above is to continue to use the CHRs, but remove all the restricting conditions we had already imposed on the definition of class and instances. As we mentioned before, the generated CHRs may no longer be terminating and confluent, but non-confluent and non-terminating systems can still be practically useful provided that we have a mechanism to prevent the resulting problems.

In this thesis we study the confluence and non-termination problem of the generated CHRs and propose solutions to deal with them. In fact, we mainly use this problem in functional dependency CHRs as a motivation to investigate confluence and termination properties in a broader context and propose some solutions that are applicable to CHR systems in general.

As we will see in Chapter 4, one solution for the confluence problem is to give priority to propagation rule applications over simplification rule applications. We will talk more about these two types of CHRs in the next chapter, but as an informal definition, simplification rules replace a subset of constraints in the constraint set with a new constraint set, but propagation rules add new constraints to the constraint set. It can be proved that if we

postpone simplification rule applications until all possible propagation rules are applied, the CHR system is always confluent.

Based on the new system of rule applications that we call *prioritized CHRs*, we will deal with the problem of non-termination in Chapter 5. The idea is to find a way to detect non-terminating rule applications in constraint solvers. In this way, rather than rejecting type class or instance definitions because they might cause non-termination, we can accept the definitions but have proper checking to reject erroneous usages of these definitions in the code.

The method presented at the end of Chapter 5 to detect infinite derivations is based on the properties of a certain rule derivation, namely *repetition candidate*. A repetition candidate is a derivation that satisfies two properties. The first condition requires the first constraint set of the derivation to *match* with the last constraint set, and the second condition involves the behavior of primitive rule applications and free variables.

We start by proving that if this kind of rule derivation occurs in the constraint solver a non-terminating sequence of rule applications can happen. The *deduction tree* definition at the end of Chapter 5 basically produces all the possible repetition candidates for a set of CHRs. So any algorithm that can build this tree structure for a set of rules can find us initial constraint sets producing repetition candidates.

The prioritized CHR system and the infinite chain properties presented in Chapter 4 are based on a formal framework for different characteristics of CHRs built in Chapter 3. As will be explained in detail, CHRs can have multi-constraint heads, variables in the tail constraints that are not used in the head (free variables), and also primitive constraints in the tail. All these characteristics will be carefully studied and mathematically formalized.

Based on this formalization we will find some important properties that are later used in our main theorems about non-termination. These properties are also interesting by themselves and can be used to study other behaviors of CHR systems. One of the properties in particular, is based on a novel approach in working with substitutions. Substitutions are widely used functions in term rewriting literature, but in our formalization we sometimes need to look at them as equational constraints. This extension of the substitution concept

results in some interesting consequences which can also have some applications in other term rewriting systems.

Before talking about the details of our new method, we will have a quick review of the Haskell type system and constraint handling rules in the next chapter. We will also show how functional dependencies can be reformulated by CHRs and what conditions they should satisfy.

Chapter 2

Background

The main goal of the next chapters is to show how we can relax some of the restrictions on the current syntax of multi-parameter type classes in Haskell and still have a confluent system that also has a mechanism to detect non-terminating inferences. But before going through the details of our new method, we will have a quick look at the current state of the Haskell type system and how it handles multi-parameter type classes and functional dependencies between types. Using Constraint Handling Rules (CHRs) as a way to formalize functional dependencies is also briefly discussed, and some of the CHR properties are explained in the more general context of *term rewriting systems*. Finally, at the end up of this chapter we will look at an alternative approach, called associate type synonyms, that can be used as an alternative to functional dependencies.

2.1 Type Classes in Haskell

Haskell type classes extend the Hindley-Milner type system to provide a uniform solution to function overloading. A type class declaration consists of class name, class parameters, members and their type signatures. Type class functions can have several implementations in class instances, differentiated by the type of instance parameters. Imagine we need different implementations for an equality function depending on the type of values that should be compared. To define this overloaded function in Haskell, first we have to define the

type class with the type signature of the member functions (here just (≡)). Different implementations of functions are next defined in different instance declarations:

```
class Eq a where
    (≡) :: a → a → Bool

instance Eq Int where
    (≡) = primEqInt

instance Eq Char where
    (≡) = primEqChar
```

Class and instance declarations can also include super-classes as contexts. If the super-class is defined for a class, it means all the instances of that class should first have an instance of it and if an instance has a super-class, that specific instance needs to have the super-class implementation. In the next example each instance of class *Ord* should first have an instance of class *Eq*:

```
class Eq a => Ord a where
    (<) :: a → a → Bool
    (≤) :: a → a → Bool
```

To support function overloading, programming languages need to have a mechanism to determine which implementation of an overloaded function should be used for each function call. In the next section we will explain how Haskell programs with single-parameter type classes handle this by translating the source code into an intermediate code with non-polymorphic functions.

2.2 Implementing Type Classes in Haskell

As discussed in [HHPJW96], one strategy to support type classes in Haskell is to translate the source code to an intermediate language while type checking

rules are applied. This intermediate code is quite similar to the original source code to support more readable error messages; but rather than having type classes, it passes some extra information to each function call to determine the instance of a function used. This extra information is built from each instance of the classes and is called a *dictionary*. A dictionary contains all the instance members and a reference to the superclass dictionary if any exists.

While the type checking rules are applied, the correct overload of a called function is determined and the corresponding dictionary is passed to the translated function. So the translated functions are not actually the same functions in the source code, they are acting as *selectors* that find the function from the dictionary passed to them.

To type check and translate the program some information about the classes, functions in classes, dictionaries and instances is needed. This data is obtained when class and instance definitions are processed and is kept in an a structure called an *environment*. The environment is divided into different sections and each step updates or uses one or more sections. Type checking and translation starts from class definitions, next instances are processed and finally the main body of the code is translated to the intermediate language.

Translating class definitions updates the environment to keep the information about the class, its super-classes and all the members and their polymorphic type signatures. Parallel to updating the environment and type checking, each function in a type class is also translated to a *selector* function that accepts a dictionary as parameter and returns the corresponding entity for it.

Translating instance declarations updates one environment section only. For each instance, a record is added to the environment which contains a reference to the dictionary and also the type signature of the instance that the dictionary belongs to. For example for the *Eq* class and its *Int* instance we defined earlier, a record like `dictEqInt` = *Eq Int* is added to the environment. The dictionary itself is built by translating the methods inside the instance, and putting them together in a data structure.

Next the main body of the code is translated by a set of rules for expressions (The same rules are also used for translating members inside instances).

For each polymorphic function call, first the types of the input parameters are determined. Using this information and data already saved in the environment, the appropriate dictionary is found and passed to the function in the translated code.

2.3 Multi-parameter Type Classes and Functional Dependencies

A natural extension to single-parameter type classes is to allow indexing class members with more than one parameter. As discussed in [PJJM97], this new extension adds to the expressiveness of type class definitions, but at the same time causes some type checking problems, and in some cases although the program is type-checked, the result type is not exactly what the programmer meant it to be. As an example of the second problem, imagine a collection class that is parameterized over the collection type and the element type:

```
class Collection c e
    empty :: c
    insert :: c → e → c
```

But we also need to somehow specify that all the elements of each collection have the same type, otherwise a function

```
insert2 xs a b = insert (insert xs a) b
```

for inserting two elements into a collection would have the type

```
insert2 :: (Collection c e1, Collection c e2) => c → e1 → e2 → c
```

which is not what we meant.

These problems necessitate a mechanism to have more control over type class parameters. Functional dependencies are one candidate for this purpose by allowing the programmer to specify dependencies between different parameters of a type class. As an informal definition, when type b is dependent

to type a, a -> b, it means that fixing a should fix b or in other words in different instances of the class, parameters a and b should not have the same value for a but different values for b. We will see one formal definition of this concept in the next chapter when we formulate functional dependencies in terms of Constraint Handling Rules. Using functional dependencies, we can fix the problem for the class defined above:

```
class Collection c e | c → e
    empty :: c
    insert :: c → e → c
```

The functional dependency c -> e means e is determined by c, or in other words we can not have two instances having the same value for parameter c but different values for e. The same insert2 function would now have the principal type:

```
insert2 :: (Collection c e) => c → e → e → c
```

2.4 CHR and Functional Dependencies

Functional dependencies can be integrated with the dictionary-based type system discussed in previous sections, but to have a sound type inference system, we need to put some limitations on how type dependencies can be used. The need to study the implications of enforcing restrictions on FDs demands a more formal definition of the functional dependency concept. This can be done by formalizing FDs in terms of Constraint Handling Rules. Using CHRs enables us to study the consequences of having functional dependencies in a type system. This formalization also can be of practical significance because it makes it possible to use constraint solvers as an integrated part of the type checking systems. The next sections explain more what Constraint Handling Rules are and how they are currently used to formalize functional dependencies.

2.4.1 Constraint Handling Rules

Constraint programming is a programming paradigm where relations between variables can be stated in the form of constraints [Wik]. Constraints are usually used in a hybrid way with other programming paradigms or as built-in subsystems, so that they define the properties of the solution and the constraint solver finds solutions that satisfy these properties. But as discussed in [Frü98], most constraint solvers have some common problems:

Being Domain Specific Constraint solvers are typically over some specific domains, such as integer, boolean or finite domains which limits their domain of applicability.

Lack of Flexibility Constraint solvers are usually hard-wired as a built-in system. This causes the application programmer to have no control over the constraint rules.

Low-Level Syntax Even those constraint solvers that allow the user to make modifications, mostly use low-level language syntax which makes it hard to work with them.

Constraint Handling Rules (CHR) as described in [Frü98] is a high level language extension that allows user-defined constraints into a host language. In this way the user can work with a high-level, easy-to-use language which is not dependent on any specific domain and can be used with any host language having its own domain.

Each user defined constraint solver consists of a set of rules which shows how a constraint set can be replaced by another equivalent constraint set (simplification rule) or can add new constraints to the system (propagation rule). Constraints in CHR systems are of two types: *predefined* or *primitive* constraints are those that will be handled by the host language constraint solver; this type of constraint is domain specific. *User-defined* or *non-primitive* constraints are defined by the user and are not dependent on any domain.

The task of a CHR constraint solver is to apply the user-defined rules to an initial set of constraints until no other rule is applicable. If the initial

constraints are (inherently) inconsistent, the constraint solver would find it out, and if not it returns the final predefined and user-defined constraint set. The CHR constraint solver is always in interaction with the host language constraint solver by feeding it with the predefined constraints and applying the results to the current user-defined constraints in each step.

To show how constraints can be defined and applied to a set of initial constraint, imagine the constraint solver consists of the rules below:

$$\begin{aligned} rule1 &: \; E\,x \Leftrightarrow C\,x \\ rule2 &: \; C\,x, D\,x\,y \Rightarrow x = y \end{aligned}$$

rule1 is a *simplification* rule that *replaces* instances of $E\ x$ with the corresponding instances of $C\ x$. We will discuss *instances* in the following chapters, but for now, constraint E' is an *instance* of E if there exists a substitution θ such that $\theta(E) = E'$.

rule2 is a *propagation* rule that *adds* an instance of predefined constraint $x = y$ to the constraint set if instances of $C\ x$ and $D\ x\ y$ (using a substitution θ) exist in the constraint set. This new predefined instance will later be applied to the constraint set by the host system.

Assuming that we have $\{E\ a, D\ a\ b\}$ as the initial constraint set, first $rule1$ can be applied to $E\ a$ and this constraint is replaced by $C\ a$. Next $C\ a$ and Dab are matched with the head of $rule2$ and $a = b$ is generated. Applying this constraint to the current constraint set results in $\{C\ a, D\ a\ a\}$

There are two basic properties that constraint solvers, or as we will see in the next section, term rewriting systems in general, are usually required to satisfy: *termination* and *confluence*. The termination property guarantees that rule applications to any initial constraint set can not continue indefinitely, and the confluence property ensures that different rule applications to an initial set will result in equivalent final constraint sets. The formal definition of confluence is explained in the next section when we talk about term rewriting systems.

In any system working with constraint solvers, defining rules in a way that satisfies these two properties is critical. In functional dependencies formalization with CHRs, this has been done by putting some limitations on the

definition of type classes and instances. In the following sections, we will explain how FDs are formalized by CHRs and what conditions are necessary to make the system confluent and terminating. As CHRs can fit in the definition of term rewriting systems, studying termination and confluence in this broader context gives us a better understanding of the limitations imposed on the definition of functional dependencies. The following sections briefly talks about the standard methods used to prove a term rewriting system is terminating and confluent.

2.4.2 CHR as a Term Rewriting System

An *abstract rewriting system* consists of a set of objects and one or more binary relations that determine the transformations between the objects in an abstract way. A *term rewriting system*(TRS) is an abstract rewriting system where the objects are first-order terms, and where the reduction relation is presented in a standard schematic format of so-called reduction rules or rewrite rules [Ter03]. As CHR systems are a specific form of term rewriting systems, in this section we briefly discuss TRS's termination and confluence properties which are also applicable to CHR systems.

For most methods used to prove the termination of a TRS the notion of a*reduction order* plays an important role. As an informal definition a reduction order on TRS terms is a well-founded order (an order that does not admit infinite descending sequence) which is closed under substitutions and contexts. The goal is to find a reduction order on terms with which for every rule in our TRS the head term is greater than the tail term. It can be proved that if such an order can be found, the TSR is terminating.

There are several approaches to find a reduction order for a TRS, but all of them have one thing in common: they check sufficient not necessary conditions for non-termination. In other words if a method does not prove that a system is terminating it does not necessarily means that it is non-terminating. Next we briefly explain three different categories that all reduction order methods fit in. As we will see in later chapters, in our new approach to deal with the problem of termination none of these methods are applicable, as we already know that our system is non-terminating and try to find an algorithm

to detect these cases. As discussed in [Ter03], three different methods to prove termination are:

Semantical Methods In this method for each function in our TRS an *interpretation* function with an ordered set as its range should be found. Using these functions, a term evaluation can be defined that maps each TRS term to a member in the ordered set. In this way, a reduction order is defined for the terms in our TRS and, as mentioned above, if for each rule the rule head is greater than the tail, our TRS is terminating. This method is applicable to prove termination for many term rewriting systems, but the downside for it is that there is no automatic way to find these interpretation functions.

Syntactical Methods In syntactical methods finding reduction orders is more mechanical. In these methods there is no need to find an interpretation function, but an arbitrary ordering on the TRS functions is chosen. Then, according to a syntactical recursive ordering rule, e.g recursive path order, all the terms in the TRS can be ordered.

Transformation Methods For many terminating systems using the above methods fail to prove termination. Another common approach is to find a termination preserving transformations for a TRS and if it can be proved that the transformed system is terminating (by using previous methods) the original system should also be terminating.

In [SDPJS07], the proof that CHRs generated from functional dependencies are terminating is based on the semantical approach. For this purpose, a weight function, which is basically an interpretation function, is defined and taking into account the restrictions imposed on the definition of type classes, it can be proved that for each generated rule, head instances are always greater than corresponding tail instances.

2.4.3 Formalizing Functional Dependencies with CHRs

CHRs can be used to formalize functional dependencies and class constraints in type class and instance definitions. This formalization not only helps to

better study the behavior of functional dependencies, but it can also be used as part of the type inference system.

As shown in [SDPJS07], a number of generation rules are used to generate a CHRs from the instance and class definitions. Two of these rules model the class-superclass dependencies and the other two model functional dependencies. For a class and an instance declaration as below:

```
class C => TC a1 ... an | fd1,...,fdm
instance C => TC t1 ... tn
```

The following CHRs should be generated:

- **The class CHRs:** For each superclass in the class definition a *propagation* rule is generated:

  $rule: \; TC\, a_1 \ldots a_n \Rightarrow C$

- **The instance CHRs:** For each instance context a *simplification* rule is generated:

  $rule: \; TC\, t_1 \ldots t_n \Leftrightarrow C$

- **The functional dependency CHRs:** For each functional dependency fd_i of the form $a_{i1} \ldots a_{ik} \rightarrow a_{i0}$, a *propagation* rule is generated:

  $rule\; TC\, a_1 \ldots a_n,\; TC\, \theta(b_1) \ldots \theta(b_n) \Rightarrow a_{i0} = b_{i0}$

  where $a_1 \ldots a_n, b_1 \ldots b_n$ are distinct type variables and :

  $$\theta(b_j) = \begin{cases} a_j & \text{if } j \in \{i_1 \ldots i_k\} \\ b_j & \text{otherwise} \end{cases}$$

- **The instance improvement CHRs:** If class instances are defined, for each functional dependency fd_i of the form $a_{i1}, \ldots, a_{ik} \rightarrow a_{i0}$, a *propagation* rule is generated:

  $rule\; TC\, \theta(b_1) \ldots \theta(b_n) \Rightarrow t_{i0} = t_{i0}$

  where $b_1 \ldots b_n$ are distinct type variables and:

  $$\theta(b_j) = \begin{cases} a_j & \text{if } j \in \{i_1 \ldots i_k\} \\ b_j & \text{otherwise} \end{cases}$$

The first two rules are quite straightforward. The first rule says if a type class has a super-class, for every instance of that class an instance for the super-class should exist. The second rule is the same but for instance contexts. If the instance has no context, the constraint would be simplified to *true* which means that if we a constraint that matches one of our instances, this constraint is satisfied. The third rule is in fact the formal definition of functional dependencies: For two instances of the same class, if they have the same type variables for the LHS of a functional dependency, the type variable for the RHS should also be the same. The last rule can also be considered as a definition of functional dependencies in a different way which says if there exists a constraint that matches an instance declaration and the variables in the left hand side of a functional dependency are the same in the constraint and instance declaration, the left hand side variable in the constraint should be the same as the one in the instance declaration.

To assure that the generated rules are terminating and confluent, class and instance definitions should satisfy a number of conditions. The first set of conditions, namely *basic conditions*, is not related to functional dependencies and only reflects the restrictions on class and instance definitions. The second set of conditions, FD rules, deal with functional dependencies. It is important to notice that these rules are *sufficient* but not *necessary* to prove termination and confluence of the generated CHRs. In other words, some class definitions can be found that do not follow these rules, their generated CHRs still have the necessary properties. Some research work has been done to ease these conditions and some alternative conditions like *Paterson Conditions* have been found that are looser, but more complex, but still they have the same above mentioned problem, i.e., they might rule out non-problematic class definitions [SDPJS07]. A summary of basic conditions and functional dependency conditions is presented next. For more detailed definitions refer to [Jon00] and [SDPJS07].

Basic Conditions:

- The Context C of a *class* and *instance declaration* can mention only type variables and in each individual class constraint CC, all type variables

are *distinct*.

- In an instance declaration instance $C \Rightarrow TC\, t_1 \ldots t_n$, at least one of the types t_i must not be a type variable.

- The instance declarations must not overlap.

FD-Conditions:

- **Consistency Condition**

  For each defined functional dependency, there should not exist two instance definitions having the same types for the LHS and different types for the RHS.

- **Coverage Condition** For each instance definition, variables in types corresponding to the functional dependency range should be a subset of variables in types corresponding to the functional dependency domain. In other words, determining the types for the domain of a functional dependency should fully determine the type in the range.

- **Bound Variable Condition**

  For each class or instance declaration, variables in the context should be a subset of the variables used in the instance.

We saw how modeling functional dependencies can help us to formalize functional dependencies, but keeping the constraint solver terminating and confluent forces us to impose some restrictions that might rule out meaningful and useful class definitions. In the following chapters, we try to find a solution to overcome this problem, but, before that, as a potential alternative to functional dependencies, associated type synonyms are discussed.

2.5 An Alternative Approach: Associated Type Synonyms

Currently, standard Haskell only supports overloading of functions via type classes. Function overloading allows indexing functions by types but there are also many cases where we need to index data constructors by types, or in other words, the ability to choose a data constructor for a type based on input parameters. Associated types serve this purpose by extending the abstraction used in type classes.

A modified version of associated types, associated type synonyms, can also be used as an alternative approach to the problem of type dependencies which is currently formulated by CHRs. This new approach is still in the early stages of development and before more research has been done it is hard to say it would be a substitute for CHRs. In the following sections we will take a quick look at how associated types and type synonyms work and, finally, we will make a comparison between CHRs and the new formulation of the type dependency problem.

2.5.1 Associated Types

To index type constructors by types, the same class structure for function overloads can be utilized. Imagine we want to have a Map structure that keeps pairs of key/values, but for some optimization purposes we would like to have different type constructors based on different possible types of *key*. We express this idea by defining a type class MapKey parameterized by the key type and also define an associated type Map as part of our class definition:

```
class MapKey k where
    data Map k v
    empty :: Map k v
    lookup :: k → Map k v → v
```

As we can see, in addition to methods in a class definition, we have an associated type parameterized with k. As an instance of this class for type *Int*,

we can have a specific data constructor for *Map*, assuming a suitable library implementing finite maps for integers, *IntMap*:

```
instance MapKey Int v where
    data Map Int v = MapInt(IntMap.Dict v)
    empty =  IntMap.emptyDict
    lookup k (MapInt d) =  IntMap.lookupDict k d
```

To integrate associated types with Haskell's class definitions, some major changes should be made to the way dictionaries are constructed, which makes the type system a lot more complex; but this complexity is hidden from the user. Another point about the associated type implementation is that the type system supporting them can still be translated into System F [CKPJ05].

The concept of typed-index types can also be modeled in a different way by functional dependencies, as for the previous example we can have:

```
class MapKey k m v | k → m
    empty :: m
    lookup :: k → m → v
```

But especially for more complex situations, using functional dependencies for this purpose has some drawbacks [CKPJM05]:

- There are cases where translating associated types to functional dependencies would result in non-confluent CHRs. This is a major problem with the current typing system for functional dependencies, but as will be explained in later chapters, this confluence problem can be solved by modifying the way rule applications are done in the constraint solver.

- Translating associate types to functional dependencies would require changing associated types to extra parameters for a type class and hence results in more complex code. But if the modifications discussed in the following chapters prove to increase the expressiveness of functional dependencies, it still might be worthwhile accepting the complexities caused by using functional dependencies.

- Defining data types in a separate module and hiding the concrete representation of the data type from the user of the module is not possible if we use FDs to model typed-indexed types.

In this section, we explained how associated types can be translated to FDs and what the problems are for this approach. In the next section associated type synonyms are introduced and we will show how they can serve the same purpose as FDs.

2.5.2 Associated Type Synonyms and Type Dependencies

In the previous section, we showed how associated types can be defined inside class declarations and how different type constructors can be introduced in each instance. As the next natural step we would allow assignment of monotypes to be used as data constructors in instance declarations. So a data type instance can have the format $S\bar{\alpha}\ \bar{\tau} = v$, where S is an associated type synonym, $\bar{\alpha}$ are type variables with corresponding type parameters in the class declaration and the $\bar{\tau}$ and v are monotypes. According to this definition $(S\ a\ =\ Int)$ would be a valid type synonym but $(S\ =\ [Int])$ is not, obviously because $[Int]$ is not a monotype. The example below illustrates how associated type synonyms can be used:

```
Class C a where
    type B a
    foo :: a → B a

instance C Bool where
    type B Bool = Int
    foo False = 0
    foo True = 1
```

In this example, the class declaration is exactly like what we had in associated types, but in the instance declaration we have assigned a primitive

type *Int* as a type synonym for a type constructor. Although associated type synonyms seem to be a trivial extension of what we already had, they raise the issue of non-syntactic type equality which requires a thorough revision of the type inference system. As for the above example, after defining this instance *B Bool* would be equal to *Int* even though they are not syntactically equal.

Handling non-syntactic equalities needs some basic changes to the type inference rules. Firstly, we have to extend typing rules for expressions to include this type of equality. Secondly, new unification rules are required for type equalities in the presence of these type functions. We also need to handle *pending equality constraints*, which are generated during expression type checking and should be resolved later using the knowledge about associate type synonyms declarations. Subsumption algorithms should also be revised to conform with type functions [CKPJ05].

Type synonyms and functional dependencies both deal with the same problem, but in different ways. Both approaches enable us to have one type depend to another. As for the above example, we can have the same concept by using functional dependencies.:

```
Class C a b | a → b where
    foo :: a → b

instance C Bool Int where
    fool False = 0
    fool True = 1
```

As can be seen above, the same dependency we had between *a* and *B a* via type synonyms is now implemented by an extra parameter *b* in the functional dependency *a->b*.

2.5.3 Comparing the Two Approaches

Associated type synonyms can be an alternative to CHRs to model functional dependencies and in some cases, defining the same type dependencies with type synonyms would give more understandable code. But associated type

synonyms have their own drawbacks; they add to the complexity of the type system and they still do not cover all the legitimate class definitions rejected by basic and FD conditions. In the following chapters, we will show how easing basic and FD conditions and dealing with the consequent confluence and non-termination problem can be another potential solution.

Chapter 3

A Formal Framework for CHRs

In this chapter we build a formal framework to model different characteristics of CHRs. Our solution for confluence and non-termination in the following chapters is based on this formalization. We have briefly discussed CHRs in the background chapter, but we need a formal definition for different elements of a CHR system.

Definition 3.0.1.

- *A* constraint, *$C\ t_1, \ldots, t_n$, is a predicate symbol, C, with terms $t_1, \ldots, t_n$ as arguments.*
- *A constraint with equality as predicate symbol is called a* primitive constraint *otherwise is a* non-primitive constraint. *We also write the primitive constraint $(=)\ t_1, t_2$ as $t_1 = t_2$.*
- *A* simplification rule, *$R:\ S_1 \Leftrightarrow S_2$, consists of two sets of constraints S_1 and S_2 with no primitive constraints in S_1.*
- *A* propagation rule, *$R:\ S_1 \Rightarrow S_2$, is a relation between two sets of constraints S_1 and S_2 with no primitive constraints in S_1. (A relation is a subset of a cartesian product, i.e., a set of tuples.)*
- *A* primitive rule *is a propagation rule with only primitive constraints in the second constraint set.*

- *A* non-primitive rule *is a rule with only non-primitive constraints in the second constraint set.*
- *A variable x is a* free variable *in rule R if it occurs only in the right constraint set.*
- *A variable x is a* bound variable *in rule R if it occurs in the left constraint set.*

Notice that in the primitive rule definition, we only allow equality as the predicate symbol. This limitation is not part of the definition of primitive rules in CHR systems, but our formalization is based on it. Although this definition of a primitive constraint is restrictive, there are still many CHR applications, including formalization of functional dependencies, that can work with it. As we can see, guarded conditions for rules are also not included and will not be part of our formalization.

3.1 Substitution, Unification, Matching

Throughout this chapter we will use substitutions and their properties to formulate different aspects of CHRs and prove properties about non-termination and confluence, so an exact definition of what it means by substitution is essential. There are different definitions for substitution depending on the subject area this term is used in [Ter03]. In our definition, substitutions are finite domain functions that map variables to terms. As we will see later in this chapter, sometimes we need to convert substitutions to finite primitive constraint sets, so the finiteness of the domain is essential.

Definition 3.1.1. *A substitution is a finite-domain function that maps variables to terms.*

To denote the composition of two substitutions α and β, we use the notation $\alpha \cdot \beta$ in which β is applied first. It can be proved that composition of substitutions is associative, but in general not commutative. The following example shows that commutativity does not always hold for substitutions.

Example: Assume we have $\alpha = \{x \mapsto a, y \mapsto b\}$ and $\beta = \{a \mapsto c\}$. For constraint $C\,x\,y$ we have: $\alpha \cdot \beta(C\,x\,y) = C\,a\,b$ and $\beta \cdot \alpha(C\,x\,y) = C\,c\,b$.

But, in some special cases, it is possible to change the order substitutions are applied without getting different results. Lemma 3.1.2 shows when this can happen. This property is used several times during the proof of our theorems. $\mathsf{Var}(t)$ is the set of all variables used in term t; $\mathsf{ran}(\alpha)$ is the range of substitution, and by $\mathsf{dom}(\alpha)$ we mean the subset of the domain where the substitution does not work as identity.

Lemma 3.1.2. *For every substitution function α and β, if*
$\mathsf{dom}(\beta) \cap \mathsf{dom}(\alpha) = \emptyset$ *and* $\mathsf{dom}(\beta) \cap \mathsf{Var}(\mathsf{ran}(\alpha)) = \emptyset$ *and* $\mathsf{Var}(\mathsf{ran}(\beta)) \cap \mathsf{dom}(\alpha) = \emptyset$,
then we have $\alpha \cdot \beta = \beta \cdot \alpha$.

Proof: If $\alpha \cdot \beta(x) = y$ and all above conditions hold, then we have $\forall s. s \in \mathsf{Var}(\beta(x)) \Rightarrow s \notin \mathsf{dom}(\alpha)$ and x is also either a member of $\mathsf{dom}(\alpha)$ or $\mathsf{dom}(\beta)$. So either $y = \alpha(x)$ or $y = \beta(x)$. If $y = \alpha(x)$ from the above conditions we know that $\forall s. s \in \mathsf{Var}(\alpha(x)) \Rightarrow s \notin \mathsf{dom}(\beta)$, so $y = \beta \cdot \alpha(x)$ and if $y = \beta(x)$ we know that $x \notin \mathsf{dom}(\alpha)$, so $y = \beta \cdot \alpha(x)$. □

A substitution can also be applied to a set of constraints, by which we mean it is applied to all the constraints in the set. In the rest of this thesis, we also use some special types of substitutions that are defined next. These terms are in fact part of the standard term rewriting rules terminology.

Definition 3.1.3. *The substitution α is called a* unifier *for terms s and t if we have $\alpha s = \alpha t$.*

Definition 3.1.4. *The substitution α is called the* most general unifier (mgu)*:*

- *For the terms s and t, if $\alpha s = \alpha t$ and for every other unifier β we have $\exists \rho.\ \beta = \rho \cdot \alpha$.*
- *For the equational constraint $t = s$, if α is an mgu for t and s.*
- *For the equational constraint set T, if for every equational constraint in T, i.e. $t = s$, α is a unifier for t and s and for every other unifier with the same property we have $\exists \beta = \rho \cdot \alpha$.*

Definition 3.1.5. *The term s is* matched *with the term t by substitution α and we write $\alpha = \mathsf{match}(s,t)$, if $\alpha s = t$ and for all β if $\beta s = t$ then $\exists \rho.\ \beta = \rho \cdot \alpha$.*

3.2 Free Variable Instantiation

To prove any properties for a CHR system with free variables, we first need to find a mathematical way to formalize them. As we know, free variables are represented the same way as bound variables in rules, but every time a rule is applied, new instances for the free variables in the tail constraints should be generated.

In our formalization, we define a substitution $\delta_{i,k}$ that maps each free variable in rule R_i to a free variable instance. We will use the second index in $\delta_{i,k}$ to generate new instances of the same free variable each time rule R_i is called. As shown in our next formal definition, $\delta_{i,k}$ substitutions with different values of k have disjoint ranges. In later sections we will bind the index k to the derivation number the rule is applied in.

Throughout the proofs for non-termination, we also need to have access to the next instance of a free variable knowing the current instance. The substitution $\beta_{i,k}$ is defined for this purpose; it gives the value of $\delta_{i,k+1}$ knowing the value of $\delta_{i,k}$. Definition 3.2.1 shows the properties $\delta_{i,k}$ and $\beta_{i,k}$ substitutions should satisfy.

Definition 3.2.1. *For every rule R_i in rule sequence $R = \langle R_1 \ldots R_n \rangle$,* instantiation substitutions *$\delta_{i,k}$ and $\beta_{i,k}$ are defined with the following properties:*

- $\forall i,k.\ i \in \{1\,..\,n\} \Rightarrow \mathsf{dom}(\delta_{i,k}) = \mathsf{FV}(R_i)$.
- $\forall i,k,x,y.\ \delta_{i,k}(x) = \delta_{i,k}(y) \Rightarrow x = y$ *($\delta_{i,k}$ is injective)*
- $\forall i,i',k,k'.\ i \in \{1\,..\,n\} \wedge i' \in \{1\,..\,n\} \wedge (i \neq i' \vee k \neq k') \Rightarrow \mathsf{ran}(\delta_{i,k}) \cap \mathsf{ran}(\delta_{i',k'}) = \emptyset$.
- $\forall i,k.\ i \in \{1\,..\,n\} \Rightarrow \delta_{i,k+1} = \beta_{i,k} \cdot \delta_{i,k}$

It is important to note that rule names in the rule sequence are in fact meta-level names referring to the actual rules. So it is possible to have

two rule names in the sequence referring to the same rule. But the same rules in a sequence generate different free variable instances, because they have different δ substitutions differentiated with the rule name indexes, and the second property of Def. 3.2.1 prevents instance name clashes. This fact is especially important for derivations and we will return back to it in later sections.

3.3 Primitive Constraints

Primitive constraint applications can also be transformed into substitution applications. Imagine we have a primitive rule $R: \quad H \Rightarrow T$ in which H is a set of constraints and T is a set of primitive constraints. If rule R is *applicable* to constraint set S with substitution α, we should first apply the same substitution to the left and right hand sides of constraints in T, which gives us new instances of primitive constraints in T. Finally, the most general unifier (mgu) of the new primitive constraints gives us a substitution that can be applied to the constraint set S.

Example: Assume that we have primitive rule $C\ x\ y, D\ [x] \Rightarrow y = [x]$ and constraint set $\{C\ a\ [b], D\ [a], E\ b\}$. The substitution α that matches the rule head to the constraint set is $\alpha = \{x \mapsto a, y \mapsto [b]\}$. As we explained before, first we apply α to $S = \{y = [x]\}$ which gives us $\alpha S = \{[b] = [a]\}$. Finally we apply the mgu of this equation $\{a = b\}$ to the original constraint set which gives us $\{C\ b\ [b], D\ [b], E\ b\}$.

As we showed in the previous example, we need to find the *mgu* of the primitive constraint set to apply the result substitution to the initial constraint set, but sometimes the *mgu* for a set does not exist. Undefined primitive constraints add to the complexity of our non-termination properties and we will see this problem in the following chapters when we define *repetition candidate* properties.

Definition 3.3.1. *A primitive constraint set T is called* defined *if* $\mathsf{mgu}(T)$ *exists.*

We will get back to the primitive rule applications later when we define

derivation steps, but before that, we study some properties related to the application of substitutions to primitive constraint sets which are later used in our proofs for non-termination and confluence.

Lemma 3.3.2. *For every defined constraint set T and variable x we have:*

$$(\mathsf{mgu}(T))x = t \Rightarrow (\mathsf{mgu}(T))t = t$$

In other words, $\mathsf{mgu}(T)$ *is idempotent.*

Proof: We know that for every constraint set T, $\mathsf{dom}(\mathsf{mgu}(T)) \cap \mathsf{Var}(\mathsf{ran}(\mathsf{mgu}(T))) = \emptyset$. So if $x \in \mathsf{dom}(\mathsf{mgu}(T))$, none of the variables in t can be in $\mathsf{dom}(\mathsf{mgu}(T))$, which means $(\mathsf{mgu}(T))t = t$. □

Lemma 3.3.3. *If for a variable x,* $\mathsf{mgu}(T)x = t$*, for every substitution θ if θT is defined, we have:*

$$(\mathsf{mgu}(\theta T))(\theta(x)) = (\mathsf{mgu}(\theta T))(\theta(t))$$

Proof: Assume that $\mathsf{mgu}(T) = \alpha$ and $\mathsf{mgu}(\theta T) = \alpha'$. We know that α' unifies constraint pairs in θT. So for each constraint pair θS_1 and θS_2 in T we have $\alpha' \cdot \theta S_1 = \alpha' \cdot \theta S_2$ which means $\alpha' \cdot \theta$ is also a unifier for T.

On the other hand, as α is the *mgu* for T we will have $\alpha' \cdot \theta = u \cdot \alpha$ for some substitution u. Also $\alpha(x) = t = \alpha(t)$ (Lemma 3), so we should have $\alpha' \cdot \theta(x) = u \cdot \alpha(x) = u(t) = u \cdot \alpha(t) = \alpha' \cdot \theta(t)$. □

Corollary 3.3.4. *For every substitution θ and primitive constraint set T, if T and θT are defined then:*

$$\mathsf{mgu}(\theta T) \cdot \theta = \mathsf{mgu}(\theta T) \cdot \theta \cdot \mathsf{mgu}(T)$$

Proof: We show that for any arbitrary variable x, we have $(\mathsf{mgu}(\theta T) \cdot \theta)(x) = (\mathsf{mgu}(\theta T) \cdot \theta \cdot \mathsf{mgu}(T))(x)$:

- If $x \notin \mathsf{dom}(\mathsf{mgu}(T))$ the equation is trivial.

- If $x \in \mathsf{dom}(\mathsf{mgu}(T))$ let $t = (\mathsf{mgu}(T))x$, so the RHS of equation equals $\mathsf{mgu}(\theta T)(\theta(t))$ and the LHS is $\mathsf{mgu}(\theta T)(\theta(x))$. According to Lemma 4, these two terms are equal. □

Transforming primitive constraints into substitutions enables us to better study how they work. As a natural extension of this concept we can also look at substitutions as primitive constraints. This new interpretation of substitutions results in an important property that will later be used in the following sections.

Definition 3.3.5. *We define a conversion of substitutions to sets of equational constraints, mapping a substitution $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ to the constraint set $\{\!\!\{ \sigma \}\!\!\} := \{x_1 = t_1, \ldots, x_n = t_n\}$.* □

As some point in our proofs for non-termination we need to know when θT is defined for a primitive constraint set T and a specific substitution θ. Next we relate θT to $\theta\{\!\!\{ \mathsf{mgu}(T) \}\!\!\}$, which makes it possible to decide about θT definedness.

Lemma 3.3.6. *If T is defined, every unifier β for $\theta\{\!\!\{ \mathsf{mgu}(T) \}\!\!\}$ is also a unifier for θT.*

Proof: First we prove $\forall x . x \in \mathsf{Var}(T) \Rightarrow \beta\cdot\theta(x) = \beta\cdot\theta\cdot\alpha(x)$. Let $\alpha = \mathsf{mgu}(T)$, then for every $x \in \mathsf{Var}(T)$ we have:

If $x \in \mathsf{dom}(\alpha)$: Let $t = \alpha(x)$ then $\beta\cdot\theta(x) = \beta\cdot\theta(t)$ and so $\beta\cdot\theta(x) = \beta\cdot\theta\cdot\alpha(x)$.

If $x \notin \mathsf{dom}(\alpha)$: We have $\alpha(x) = x$ so $\beta \cdot \theta(x) = \beta \cdot \theta \cdot \alpha(x)$.

Next we show that β is also a unifier for θT. We know that for every constraint pair in T, $\alpha(T_1) = \alpha(T_2)$, so $\beta \cdot \theta \cdot \alpha(T_1) = \beta \cdot \theta \cdot \alpha(T_2)$ and from the equation we found for substitutions we will finally have $\beta\cdot\theta(T_1) = \beta\cdot\theta(T_2)$ which means β is also a unifier for $\theta(T)$.

Theorem 3.3.7. *For every primitive constraint set T and substitution θ we have:*

$$\mathsf{mgu}(\theta\{\!\!\{ \mathsf{mgu}(T) \}\!\!\}) = \mathsf{mgu}(\theta T)$$

Proof: From Lemma 3.3.6 we know that every unifier for $\mathsf{mgu}(\theta\{\!\!|\, \mathsf{mgu}(T) \,|\!\!\})$ is also a unifier for $\mathsf{mgu}(\theta T)$. So we only need to show every unifier for $\mathsf{mgu}(\theta T)$ is also a unifier for $\mathsf{mgu}(\theta\{\!\!|\, \mathsf{mgu}(T) \,|\!\!\})$. From Lemma 3.3.3 we know that if $\mathsf{mgu}(T)x = t$ for a variable x, we have $(\mathsf{mgu}(\theta T))(\theta(x)) = (\mathsf{mgu}(\theta T))(\theta(t))$. This means $\mathsf{mgu}(\theta T)$ is the unifier for the set of equational constraints $\theta(x) = \theta(t)$ or in other words for $\mathsf{mgu}(\theta\{\!\!|\, \mathsf{mgu}(T) \,|\!\!\})$.

Corollary 3.3.8. *If $\theta\{\!\!|\, \mathsf{mgu}(T) \,|\!\!\}$ is defined then θT is defined.*

Proof: Trivial from Theorem 3.3.7.

The next theorem show a new property for primitive constraints which is an essential part of our confluence proof in the next chapter.

Theorem 3.3.9. *For every two primitive constraint sets Q_1 and Q_2 we have:*

$$\mathsf{mgu}(\mathsf{mgu}(Q_1)Q_2) \cdot \mathsf{mgu}(Q_1) = \mathsf{mgu}(\mathsf{mgu}(Q_2)Q_1) \cdot \mathsf{mgu}(Q_2)$$

Proof: We prove that both sides of the equation are the most general unifier for $Q_1 \cup Q_2$. Starting from the lhs, as $\mathsf{mgu}(Q_1 \cup Q_2)$ is also a unifier for Q_1, for some α we should have $\mathsf{mgu}(Q_1 \cup Q_2) = \alpha \cdot \mathsf{mgu}(Q_1)$. So $\mathsf{mgu}(Q_1)$ is first applied to $Q_1 \cup Q_2$ which unifies Q_1. So we only need the most general unifier for $\mathsf{mgu}(Q_1)Q_2$ which is $\mathsf{mgu}(\mathsf{mgu}(Q_1)Q_2)$. This means $\alpha = \mathsf{mgu}(\mathsf{mgu}(Q_1)Q_2)$ and $\mathsf{mgu}(Q_1 \cup Q_2) = \mathsf{mgu}(\mathsf{mgu}(Q_1)Q_2) \cdot \mathsf{mgu}(Q_1)$. The same argument can be repeated for the rhs.

3.4 Propagation Rule Applications and Derivations

In this section we clarify what we mean by a sequence of rule applications. We start by defining the building block of a rule sequence, *derivation step.* A derivation step corresponds to the definition of *reduction* in the abstract term rewriting rules context [Ter03]. We give a formal definition for this term based on our formalizations of primitive and non-primitive rules.

Definition 3.4.1. *For the rule $R_i: \ H_i \Rightarrow T_i$, constraint sets S, S', Q, and an arbitrary number k, we define two kinds of* derivation step*:*

- $S \xrightarrow[k]{R_i} S'$ *is a non-primitive derivation, if R_i is a non-primitive rule and we have:*

  $\exists \alpha, Q.\ (Q \subseteq S) \ \wedge \ (\alpha = \mathsf{match}(H_i, Q)) \ \wedge \ (S' = S \cup (\alpha \cdot \delta_{i,k})(T_i))$

- $S \xrightarrow[k]{R_i} S'$ *is a primitive derivation step, if R_i is a primitive rule and we have:*

  $\exists \alpha, Q.\ (Q \subseteq S) \ \wedge \ (\alpha = \mathsf{match}(H_i, Q)) \ \wedge \ (S' = (\mathsf{mgu}((\alpha \cdot \delta_{i,k})T_i))S)$

A derivation step $S \xrightarrow[k]{R_i} S'$ is called trivial *if $S = S'$, and* non-trivial *otherwise.*

In [SDPJS07], when operational semantics of CHRs are discussed, different types of derivation steps are also explained. But some differences between that formulation and our approach are observable:

- Here we have merged the *solve step* with the primitive derivation step. In fact, the solve step presented in [SDPJS07] is always done implicitly when primitive rules are applied. Furthermore having only two main steps makes our future theorems easier to present.

- In our primitive derivation we do not keep the primitive constraints after they are applied to the constraint set. The idea is that when primitive constraints are applied, the updated constraints can never generate new primitive constraints that are inconsistent with the current ones. This is obvious because every time a primitive constraint is applied all the variables in the domain of *mgu* are substituted and no longer exist. So the already applied primitive constraints are in fact never used again.

- Derivation steps in [SDPJS07] are based on the fact that every time a rule is applied, all the variables are renamed. But in the following theorems we will need a more sophisticated way to formulate the definition of free variables.

Based on the definition of the derivation step, now we can define what we mean exactly by a sequence of rule applications or a *derivation.*

Definition 3.4.2. *A derivation D_k : $S_1 \xrightarrow[k]{R_1} S_2 \xrightarrow[k]{R_2} \ldots$, with the derivation number k, is a sequence of primitive or non-primitive derivation steps in which every middle rule is applied on the result constraint set of the previous derivation step. $\langle R_1, R_2 \ldots \rangle$ is the* rule sequence *for derivation D_k. A derivation with infinite derivation steps is called an* infinite derivation. *For derivation D_k, we define:*

- *$R_{i,k}$ is the application of rule R_i in derivation D_k.*
- *$\alpha_{i,k}$ is the substitution that matches the lhs of R_i to the subset of S_i in derivation D_k underlying the application of R_i.*
- *$H_{i,k}$ is the head instance of rule R_i after application of $\alpha_{i,k}$.*
- *$T_{i,k} := \alpha_{i,k} \cdot \delta_{i,k} T_i$.*

As we discussed before, $R_1, \ldots, R_n$ are meta-level names referring to the rules. So we can have different R_i's referring to the same rule.

Based on the definition of derivation we can now define what we mean by a chain of derivations.

Definition 3.4.3. *A chain N_n : $D_1 \ldots D_n$ is a sequence of derivations with the same rule sequence $R_1 \ldots R_k$ where the first constraint set of each sequence is equal to the last constraint set of the previous derivation.*

Chapter 4

The Confluence Problem

The current approach to guarantee confluent CHRs for functional dependencies imposes the Basic and FD conditions on instance and class definitions to exclude cases that might cause non-confluent (or non-terminating) rules. Our new approach relaxes the conditions by keeping only those that are inherent in the definition of functional dependency and deals with the confluence problem by focusing on the behavior of CHRs and constraint solvers to find where the problem initially stems from and what can be done to prevent it.

4.1 Problem Description

In a non-confluent system, different orders of applying CHRs to constraints can result in different sets of constraints that are syntactically unjoinable, but the logical meanings of the constraint sets are the same [Abd97]. (Logical equivalence exists only if the rules are well defined, i.e., they are consistent and complete. We consider this case later in example 3.)

As we will see, it can be proved that non-confluence happens due to applying simplification rules, but as an informal argument, when we simplify a subset of the constraint set, we are removing it from our constraint set and hence we might limit our choices for the next rules to apply, so in the presence of simplification rules the order of rule application might matter. The interesting observation is that although different constraint sets generated by

non-confluent CHR systems are logically equivalent but some constraint sets contain more information for the client using the constraints. As an example consider the rules:

$D \Leftrightarrow C$

$D \Leftrightarrow A \wedge B \wedge C$

The above CHRs are non-confluent as we can have C or $A \wedge B \wedge C$ as the final states, by applying the first or the second rule to constraint D. As can be seen the final two sets are logically equivalent but the second set gives us more information about our initial goal set.

Even though CHR theory rejects the non-confluent systems, these systems can be of practical importance for certain applications, especially if the constraint set containing more information is obtainable. By making some changes to the existing CHRs and the way CHRs are applied to constraint sets by the constraint solver, we can change a non-confluent system into a confluent one which provides the most complete final set and in some cases reveals inconsistencies that are hidden in other constraint sets.

4.2 Prioritized CHRs

Our solution for this problem is based on giving priority to propagation rules over simplification rules. To do so we use this fact that applying a simplification rule $H \Leftrightarrow T$, has the same effect as applying the simplification rule $H \Leftrightarrow true$ after application of propagation rule $H \Rightarrow T$.

Definition 4.2.1. *A prioritized CHR system is the combination of two CHR systems that are executed consecutively, the first one with only propagation rules and the second one with only simplification rules.*

---

i) Replace every simplification rule $H \Leftrightarrow T$ with $H \Rightarrow T$ and $H \Leftrightarrow true$.

ii) Build the first CHR system with all the propagation rules of the modified CHR system in step 1.

iii) Build the second CHR system with all the simplification rules of the modified CHR system in step 1.

Changing a CHR System to a Prioritized CHR System

The reason behind this approach is that by postponing the application of simplification rules we do not lose any constraints that might match with other rules.

Using this method for the above example gives us $A \wedge B \wedge C \wedge D$ after applying propagation rules in the first CHR system and $A \wedge B \wedge C$ after applying the simplification rules in the second CHR system, so the final result is the one we are interested in: a set of constraints having all the information derivable from the rules.

Using this method to generate CHRs from functional dependencies yields some interesting results that will be discussed next; but before looking at some examples we revise the three FD conditions to see which are really necessary and which are defined just to avoid non-confluence and non-termination.

The first FD condition, the consistency condition, rules out inconsistent conditions and is in fact another definition for what we mean by functional dependencies. But the next two conditions are apparently defined to prevent non-confluent and non-terminating CHRs. Currently we focus on the confluence problem and postpone the termination problem to be dealt with in later chapters. So the ideal is to have only the consistency condition for instance declarations and our system still be terminating and confluent. In the next section we will study some examples to see how giving priority to propagation rules can solve the confluence problem.

4.3 Prioritized CHRs by Examples

In this section we study some class and instance definitions that violate FD conditions and are not confluent. For each example we will see how the new

method of CHR application can solve the problem.

4.3.1 Example 1 Coverage Condition

Consider the following class and instance declarations that violate the coverage condition (*b* is not a free variable in *[a]*):

```
class C a b | a -> b
instance D b => C [a] (a,b)
```

The CHRs generated from the above declaration with the new method are:

The first CHR system with propagation rules:

```
rule C [a] (a,b) ==> D b
rule C a b1 , C a b2 ==> b1 = b2
rule C [a] b ==> b = (a,b1)
```

The second CHR system with simplification rules:

```
rule C [a] (a,b) <==> true
```

Examine the two constraints: *C [c] d1* and *C [c] d2*. After applying propagation rules we would have

```
C [c] (c,b1) , C [c] (c,b2),
D b1, D b2, d1 = d2 ,
d1 = (c,b1), d2 = (c,b2), b1 = b2.
```

And finally by applying simplification rule we find that

```
d1 = d2 = (c,b1), D b1
```

that completely conforms to the functional dependency and instance declaration we had in the program.

4.3.2 Example 2 Consistency Condition

Consider the inconsistent instance declarations below:

```
class C a b | a → b
instance c [a] (Maybe a)
instance c [b] b
```

Above declarations would generate the following rules:

The first CHR system with propagation rules:

```
C a b1, C a b1 ==> b1 = b2
C [a] a1 ==> a1 = Maybe a
C [b] b1 ==> b1 = b
C [a] (Maybe a) ==> true
C [a] a ==> true
```

The second CHR system with simplification rules:

```
C [a] (Maybe a) <==> true
C [a] a <==> true
```

For the two constraints *C [a] b1* and *C [a] b2*, we would have:

```
C [a] b1 , C [a] b2 ,
b1 = b2, b1 = Maybe a ,
b1 = a, b2 = Maybe a, b2 = a
```

The result is clearly inconsistent because *a* and *Maybe a* are not unifiable. This was expected as we violated the consistency rule. Example 3 shows how we can detect problems like this in our instance declarations by utilizing CHRs rules.

4.3.3 Example 3

Consider the following class and instance declarations:

```
class D a b
class D a b => C a b
instance C [a] a
```

Here are the CHRs arising from the above declarations:

```
C [a] a <==> True
C a b ==> D a b
```

Here we have both basic conditions and Jones's FD conditions satisfied, but the program logically implies that we should have an instance declaration for *D[a] a*. As the CHR generation rules assume that the instance and class declarations are correctly defined, the resulting CHRs are non-confluent (Consider *D[a] a*; applying the rules above can either generate ***true*** or *D[a] a* that are not only non-joinable but also logically inequivalent). As the current formulation of CHRs are theoretically useful but in practice there is no guarantee that the programmer has defined correct and consistent class and instance declarations, so a mechanism to check declarations is necessary. Currently part of this is done by Basic and FD conditions but some other cases like the above example should be taken care of separately. Here we propose a method to utilize the generated CHRs themselves to detect such errors in them. Below is the general procedure to check the declaration part of a program:

---

i) Build an initial constraint set from instance declarations by adding the constraint *TC* `t1..tn` for every instance declaration **`instance`** *T* => *TC* `t1..tn`.

ii) Apply the propagation rules of the first CHR system. If the rule application fails it mean there are problems in the class and instance definitions (consistency violation, for example).

iii) Apply simplification rules of the second CHR system. If there is any constraint rule left in the constraint set after the rule applications, it means some instance declarations are missing in our declarations.

Detecting Unsound Rule Definitions

Applying the above procedure to our example would result in *D[a] a*, that can not be simplified to *true*, meaning that an instance declaration for this class D is missing.

4.4 Prioritized CHR and Confluence

In this section we prove why prioritized CHR systems are deterministic and always give us the same final result set for any initial constraint set.

Lemma 4.4.1. *A CHR system with only propagation rules is confluent.*

Proof: We prove confluence via the diamond property, namely we show that for every rule R_1 and R_2 and constraint sets S, S_1 and S_2 if $S \xrightarrow{R_1} S_1$ and $S \xrightarrow{R_2} S_2$, then there exists a constraint set S' where $S_1 \xrightarrow{R_2} S'$ and $S_2 \xrightarrow{R_1} S'$.

If R_1 and R_2 are non-primitive: From Def. 3.4.1 $S_1 = S \cup \alpha_1(T_1)$ and $S_2 = S \cup \alpha_2(T_2)$ for some substitutions α_1 and α_2. Applying R_2 to S_1 and R_1 to S_2 gives the same constraint set $S' = S \cup \alpha_1(T_1) \cup \alpha_2(T_2)$.

If R_1 is primitive, R_2 is non-primitive: From Def. 3.4.1 $S_1 = \theta(S)$ and $S_2 = S \cup \alpha(T_2)$ for some substitutions θ and α. Applying R_2 to S_1 and R_1 to S_2 gives the same constraint set $S' = \theta(S) \cup \theta \cdot \alpha(T_2)$.

If R_1 and R_2 are primitive: From Def. 3.4.1 $S_1 = \mathsf{mgu}(\alpha_1(T_1))S$ and $S_2 = \mathsf{mgu}(\alpha_2(T_2))S$ for some substitutions α_1 and α_2. We need to prove the result of applying R_2 to S_1 is the same result of applying R_1 to S_2, that is:

$\mathsf{mgu}(\mathsf{mgu}(\alpha_1(T_1))\cdot\alpha_2(T_2))\cdot\mathsf{mgu}(\alpha_1(T_1))S = \mathsf{mgu}(\mathsf{mgu}(\alpha_2(T_2))\cdot\alpha_1(T_1))\cdot \mathsf{mgu}(\alpha_2(T_2))S$

Using Theorem 3.3.9 ($Q_1 = \alpha_1(T_1)$ and $Q_2 = \alpha_2(T_2)$) the equation holds.

Theorem 4.4.2. *Reductions to normal form using a prioritized CHR system are deterministic.*

Proof: As the first CHR system of a prioritized CHR system contains only propagation rules and according to Lemma 4.4.1 is confluent, reductions to normal form using the first system is deterministic. Confluence of the second CHR system is also trivial and so the following reductions done by the second system are also deterministic. This means the reductions done by the whole system are deterministic.

Chapter 5

The Non-Termination Problem

Type class and instance declarations with no FD restrictions imposed, can result in CHRs that are not terminating. In this chapter we attempt to find a way to detect these non-terminations in the constraint solver. Using this method enables us to still work with CHRs that are non-terminating in general, but only cause non-termination in certain situations and in this way incorporate more class definitions as valid. Before showing how to detect non-termination cases, we introduce a special type of derivation and prove that if this derivation happens during rule applications, the constraint solver never terminates.

All the findings of this chapter are based on the formalized framework we constructed in Chapter 3. We also assume that the constraint solver is prioritized, so we only need to focus on propagation rule applications, because the last step of applying simplification rules would never cause non-termination. (Recall that all new simplification rules simplify a constraint to *true*.)

5.1 Matching Constraints and Infinite Derivations

The method to detect non-termination presented at the end of this chapter is based on the theory we will introduce next. The first lemma relates two different derivation steps with the same rule. This lemma is in fact the building block of our main theorem.

Lemma 5.1.1. *Assume that we have a derivation step, $S_1 \xrightarrow[k]{R_i} S_2$ for the rule $R_i : H_i \Rightarrow T_i$. For a substitution θ let $S'_1 = \theta S_1$, then we have:*

i) $\exists \alpha' \cdot \alpha'(H_i) \subseteq S'_1$.

ii) We can have a derivation step $S'_1 \xrightarrow[k+1]{R_i} S'_2$ such that $\exists \theta' \cdot \theta' S_2 = S'_2$ (provided $T_{i,k+1}$ is defined if R_i is primitive).

Proof: (1) From the derivation step $S_1 \xrightarrow[k]{R_i} S_2$ we know that $\exists \alpha.\alpha(H_i) \subseteq S_1$. So from $\alpha(H_i) \subseteq S_1$ and $\theta S_1 = S'_1$ we can find $\alpha' = \theta \cdot \alpha$ that satisfies the predicate $\alpha'(H_i) \subseteq S'_1$.

(2) From (1) and the lemma assumption that $T_{i,k+1}$ is defined (if R_i is primitive), we know that R_i can be applied to S'_1 and so it is possible to have a derivation step $S'_1 \xrightarrow[k+1]{R_i} S'_2$. To find θ' and S'_2 we consider two cases:

R_i is non-primitive Starting from S'_2:

$$
\begin{aligned}
S'_2 &= S'_1 \cup (\alpha' \cdot \delta_{i,k+1})T_i && \text{Def. 3.4.1} \\
&= \theta S_1 \cup (\alpha' \cdot \delta_{i,k+1})T_i && \theta S_1 = S'_1 \\
&= \theta S_1 \cup (\theta \cdot \alpha \cdot \delta_{i,k+1})T_i && \alpha' = \theta \cdot \alpha \\
&= \theta S_1 \cup (\theta \cdot \alpha \cdot \beta_{i,k} \cdot \delta_{i,k})T_i && \text{Def. 3.2.1} \\
&= \theta S_1 \cup (\theta \cdot \beta_{i,k} \cdot \alpha \cdot \delta_{i,k})T_i && \text{Lemma 3.1.2} \\
&= \theta S_1 \cup (\theta \cdot \beta_{i,k})(S_2 - S_1) && (\alpha \cdot \delta_{i,k})T_i = S_2 - S_1
\end{aligned}
$$

Also we know that $\beta_{i,k}$ has no effect on S_1, because we can not have the k_{th} instances of the R_i free variables in S_1, so we have $S_1 = \beta_{i,k} S_1$. Using this equation:

$\theta S_1 \cup (\theta \cdot \beta_{i,k})(S_2 - S_1) = (\theta \cdot \beta_{i,k})S_1 \cup (\theta \cdot \beta_{i,k})(S_2 - S_1) = (\theta \cdot \beta_{i,k})(S_1 \cup (S_2 - S_1)) = (\theta \cdot \beta_{i,k})(S_1 \cup S_2)$.

Also from Def. 3.4.1 we know that $S_2 = S_1 \cup (\alpha \cdot \delta_{i,k})H_i$ so $S_1 \cup S_2 = S_2$ and finally we will have $S'_2 = (\theta \cdot \beta_{i,k})S_2$ which means:

$$\theta' = \theta \cdot \beta_{i,k}$$

R_i is primitive Starting from S'_2:

$$\begin{aligned} S'_2 &= \mathsf{mgu}((\alpha' \cdot \delta_{i,k+1})T_i)S'_1 && \text{Def. 3.4.1} \\ &= (\mathsf{mgu}((\alpha' \cdot \delta_{i,k+1})T_i) \cdot \theta)S_1 && S'_1 = \theta S_1 \\ &= (\mathsf{mgu}((\theta \cdot \alpha \cdot \delta_{i,k+1})T_i) \cdot \theta)S_1 && \alpha' = \theta.\alpha \\ &= (\mathsf{mgu}((\theta \cdot \alpha \cdot \beta_{i,k} \cdot \delta_{i,k})T_i) \cdot \theta)S_1 && \text{Def. 3.2.1} \\ &= (\mathsf{mgu}((\theta \cdot \beta_{i,k} \cdot \alpha \cdot \delta_{i,k})T_i) \cdot \theta)S_1 && \text{Lemma 3.1.2} \\ &= (\mathsf{mgu}((\theta \cdot \beta_{i,k})((\alpha \cdot \delta_{i,k})T_i)) \cdot \theta)S_1 \end{aligned}$$

Also for the same reason as the non-primitive part of proof, we know $S_1 = \beta_{i,k}S_1$, so:

$(\mathsf{mgu}((\theta \cdot \beta_{i,k})((\alpha \cdot \delta_{i,k})T_i)) \cdot \theta)S_1 = (\mathsf{mgu}((\theta \cdot \beta_{i,k})((\alpha \cdot \delta_{i,k})T_i)) \cdot \theta \cdot \beta_{i,k})S_1$

Now we can use Corollary 3.3.4. We already know that $(\alpha \cdot \delta_{i,k})T$ is defined because we have the derivation step $S_1 \xrightarrow[k]{R_i} S_2$, also from the assumptions we know that $T_{i,k+1} = (\theta \cdot \beta_{i,k})((\alpha \cdot \delta_{i,k})T_i)$ is defined, so we can apply Corollary 3.3.4 that gives us:

$$\begin{aligned} & (\mathsf{mgu}((\theta \cdot \beta_{i,k})((\alpha \cdot \delta_{i,k})T_i)) \cdot \theta \cdot \beta_{i,k})S_1 \\ = \; & (\mathsf{mgu}((\theta \cdot \beta_{i,k})((\alpha \cdot \delta_{i,k})T_i)) \cdot \theta \cdot \beta_{i,k} \cdot \mathsf{mgu}((\alpha \cdot \delta_{i,k})T_i))S_1 && \text{Corollary 3.3.4} \\ = \; & (\mathsf{mgu}((\theta \cdot \beta_{i,k})((\alpha \cdot \delta_{i,k})T_i)) \cdot \theta \cdot \beta_{i,k})S_2 && \text{Def. 3.2.1} \end{aligned}$$

We reached $\mathsf{mgu}((\theta \cdot \beta_{i,k})((\alpha \cdot \delta_{i,k})T_i))$ from $\mathsf{mgu}((\alpha' \cdot \delta_{i,k+1})T_i)$, so by replacing it in above equation we have:

$S'_2 = (\mathsf{mgu}((\alpha' \cdot \delta_{i,k+1})T_i) \cdot \theta \cdot \beta_{i,k})S_2 = (\mathsf{mgu}(T_{i,k+1}) \cdot \theta \cdot \beta_{i,k})S_2$

Which gives us θ' for the primitive rule case:

$$\theta' = \mathsf{mgu}(T_{i,k+1}) \cdot \theta \cdot \beta_{i,k} \quad \square$$

From the results of Lemma 5.1.1 we can find a relation between θ substitutions of different derivations. Assume that a chain has $\langle R_1, \ldots, R_n \rangle$ rule sequence. We define rule sequence $\langle M_1, \ldots, M_m \rangle$ as the subsequence $\langle R_1, \ldots, R_n \rangle$ with only primitive rules and if M_i in this sequence corresponds to R_j in $\langle R_1, \ldots, R_n \rangle$, we use $M_{i,k}$ to refer to $T_{j,k}$ and $\mu_{i,k}$ to refer to $\mathsf{mgu}(T_{j,k})$.

We define $\theta_{i,k}$ as the substitution that maps the i_{th} constraint set in derivation D_k to the i_{th} constraint set in derivation D_{k+1} and we intend to find a relation between $\theta_{i,k}$ and $\theta_{i,k+1}$. We also assume that M_g is the first primitive rule after the rule R_{i-1} (if any primitive rules exist after R_{i-1}) and M_l is the first primitive rule before the rule R_i (if any primitive rules exist before the rule R_i).

From Lemma 5.1.1 we know that $\theta_{i+1,k} = \theta_{i,k} \cdot \beta_{i,k}$ if the rule is non-primitive and $\theta_{i+1,k} = T_{i,k+1} \cdot \theta_{i,k} \cdot \beta_{i,k}$. So we have:

$$\theta_{n+1,k-1} = \mu_{m,k} \ldots \mu_{g,k} \cdot \theta_{i,k-1} \cdot \beta_{i,k-1} \ldots \beta_{n,k-1}$$

$$\theta_{i,k} = \mu_{l,k+1} \ldots \mu_{1,k+1} \cdot \theta_{1,k} \cdot \beta_{1,k} \ldots \beta_{i-1,k}.$$

On the other hand from the definition of chain we know that $\theta_{1,k} = \theta_{n+1,k-1}$. So substituting the first equation in the second one gives us:

$$\theta_{i,k} = \mu_{l,k+1} \ldots \mu_{1,k+1} \cdot \mu_{m,k} \ldots \mu_{g,k} \cdot \theta_{i,k-1} \cdot \beta_{i,k-1} \ldots \beta_{n,k-1} \cdot \beta_{1,k} \ldots \beta_{i-1,k}$$

Next we introduce a special kind of derivations, *repetition candidates*, and will show how these derivations can repeat themselves. Informally it means if we start with the last constraint set of a repetition candidate, we can apply the same rule sequence of the first derivation and have another repetition candidate. Next definition shows what exactly it means by a repetition candidate.

Definition 5.1.2. *Derivation* $D_k : S_1 \xrightarrow[k]{R_1} S_2 \ldots \xrightarrow[k]{R_n} S_{n+1}$ *in chain* N_k *is a* repetition candidate *if:*

i) $\exists \theta . \theta(S_1) \subseteq S_{n+1}$.

ii) For all i $(1 \leq i \leq n)$, *if* R_i *is a primitive rule:*

$$\forall c \in \mathsf{dom}(\mathsf{mgu}(T_{i,k})).(\exists i', k'.\ c \in \mathsf{ran}(\delta_{i',k'}) \wedge ((k' = k \wedge i' < i) \vee (k' = k-1 \wedge i' \geq i)))$$

which means all the members of $\mathsf{dom}(\mathsf{mgu}(T_{i,k}))$ *are the latest free variable instances that existed before the application of* $\xrightarrow[k]{R_i}$

iii) *For all i, j $(1 \leq i \leq n)$, if R_i and R_j are primitive rules (i and i' can be equal):*

$$\forall j, j', k', k'' \; .$$
$$\forall c \in \mathsf{dom}(\mathsf{mgu}(T_{i,k})) \cap \mathsf{ran}(\delta_{j,k'}) \; .$$
$$\forall c' \in \mathsf{dom}(\mathsf{mgu}(T_{i',k})) \cap \mathsf{ran}(\delta_{j',k''}) \; . \; c \neq c' \Rightarrow (\delta^{-1}_{j,k'}(c) \neq \delta^{-1}_{j',k''}(c') \vee j \neq j')$$

which means that free variable instances in different $\mathsf{dom}(\mathsf{mgu}(T_{i,k}))$*s, are from different free variables.*

At first glance, it might seem that if (ii) is true, (iii) is automatically satisfied, but there are some cases that we also need the third condition. Assume that the rule R_i has free variable v. In derivation D_k we can have $\delta_{i,k-1}(v)$ instantiated before the application of R_i, and $\delta_{i,k}(c)$ instantiated after the application of R_i. In this case the second condition is satisfied but the third condition rejects that, because in the domain of two primitive rule applications in a derivation we can not have two variable instances from the same free variable.

We also have an extra condition $\delta^{-1}_{j,k'}(c) \neq \delta^{-1}_{j',k''}(c')$ to allow cases that both free variable instances are coming from the same rule but are from different free variables. But we can not use this condition when c and c' are coming from different rules, because our definition of derivation allows having a rule appear more than once in a derivation. For this reason, we have $j \neq j'$ to allow c and c' both have the same free variable but come from different positions j and j' in a derivation.

The next theorem uses these properties to show how repetition candidates can be repeated in a chain.

Theorem 5.1.3. *If derivation*

$$D_k : S_1 \xrightarrow[k]{R_1} S_2 \xrightarrow[k]{R_2} \ldots \xrightarrow[k]{R_n} S_{n+1}$$

is a repetition candidate in chain N_k, then chain N_{k+1} also exists and the last derivation of N_{k+1}(i.e. D_{k+1}) is a repetition candidate.

Proof: We need to prove that the derivation $D_{k+1} : S'_1 \xrightarrow[k]{R_1} S'_2 \xrightarrow[k]{R_2} \ldots \xrightarrow[k]{R_n} S'_{n+1}$

with $S'_1 = S_{n+1}$ exists and is a repetition candidate. We prove by induction on derivation steps of D_{k+1}.

For the induction step, we prove that if $S'_1 \xrightarrow[k]{R_1} \ldots S'_i$ exists and the following conditions hold:

i) $\theta_{i,k} S_i = S'_i$ for a substitution $\theta_{i,k}$.

ii) For every substitution $\mu_{j,k+1}$, where M_j is a primitive rule before R_i in rule sequence we have: $\forall c \in \mathsf{dom}(\mu_{j,k+1}).\exists \beta, c' \in \mathsf{dom}(\mu_{j,k}).\ \beta c' = c$ where β is an instantiation substitution (it means all the members of $\mathsf{dom}(\mu_{j,k+1})$ are the next instances of $\mathsf{dom}(\mu_{j,k})$).

then $R_{i,k+1}$ can be applied to S'_i and the above conditions hold this time for S_{i+1} (replacing all i's with $i+1$'s in above conditions).

If R_i is non-primitive, from Lemma 5.1.1 and the first condition, we know that $R_{i,k+1}$ is applicable and we also have $\theta_{i+1,k} S_{i+1} = S'_{i+1}$. The second condition also holds after the application of R_i because we still have the same primitive rule applications in D_{k+1}.

If R_i is primitive, according to Lemma 5.1.1, $R_{i,k+1}$ is applicable to S'_i if $T_{i,k+1}$ is defined. We know that $(\theta_{i,k} \cdot \alpha_{i,k}) T_i = T_{i,k+1}$. From Theorem 3.3.7, to prove that $\mathsf{mgu}((\theta_{i,k} \cdot \alpha_{i,k}) T_i)$ exists, we can prove that $\mathsf{mgu}(\theta_{i,k} \{\!\!\{ \mathsf{mgu}(\alpha_{i,k} T_i) \}\!\!\}) = \mathsf{mgu}(\theta_{i,k} \{\!\!\{ \mathsf{mgu}(T_{i,k}) \}\!\!\})$ exists. First we look at the result of $\theta_{i,k} \{\!\!\{ \mathsf{mgu}(T_{i,k}) \}\!\!\}$.

We substitute $\theta_{i,k}$ with its equivalent we found in terms of $\theta_{i,k-1}$. So we have:

$\theta_{i,k} \{\!\!\{ \mathsf{mgu}(T_{i,k}) \}\!\!\} = \mu_{l,k+1} \ldots \mu_{1,k+1} \cdot \mu_{m,k} \ldots \mu_{g,k} \cdot \theta_{i,k-1} \cdot \beta_{i,k-1} \ldots \beta_{n,k-1} \cdot \beta_{1,k} \ldots \beta_{i-1,k} \{\!\!\{ \mathsf{mgu}(T_{i,k}) \}\!\!\}$

where $\mu_{m,k}$ is the last primitive rule application in D_k and $\mu_{l,k+1}$ the one immediately before the application of $R_{i,k+1}$ and $\mu_{g,k}$ equals to $\mathsf{mgu}(T_{i,k})$ (because R_i is primitive).

From Def. 5.1.2.ii) we know that for every $c \in \mathsf{dom}(\mathsf{mgu}(T_{i,k}))$, we have $c = \delta_{i_c,k_c}(v)$ for some free variable v and for some i_c and k_c satisfying $((k_c = k \wedge i_c < i) \vee (k_c = k-1 \wedge i_c \geq i))$. We study the effect of $\mu_{l,k+1} \ldots \mu_{1,k+1} \cdot \mu_{m,k} \ldots \mu_{g,k} \cdot \theta_{i,k-1} \cdot \beta_{i,k-1} \ldots \beta_{n,k-1} \cdot \beta_{1,k} \ldots \beta_{i-1,k}$ on c (as we mentioned $c \in \mathsf{dom}(\mathsf{mgu}(T_{i,k}))$).

If $k_c = k \wedge i_c < i$: Substitution $\beta_{i_c,k}$ in $\beta_{1,k} \dots \beta_{i-1,k}$ changes c to $\beta_{i_c,k}(c)$. Substitution $\theta_{i,k-1}$ has no effect on $\beta_{i_c,k}(c)$ because members of $\mathsf{dom}(\theta_{i,k-1})$ are variables in D_{k-1} and can not be equal to $\beta_{i_c,k}(c)$.

$\mu_{m,k} \dots \mu_{g,k}$ also has no effect on $\beta_{i_c,k}(c)$ because domain members of each of these substitutions are variables in D_k and can not be equal to $\beta_{i_c,k}(c)$.

Also, $\beta_{i_c,k}(c)$ can not be in domain of any substitution in $\mu_{1,k+1} \dots \mu_{l,k+1}$, because from (ii) in the induction step conditions, c should be in domain of a substitution in $\mu_{1,k} \dots \mu_{l,k}$, which is not possible, because if c is in the domain, it has already been substituted by a term and can not reappear in $\mathsf{dom}(\mathsf{mgu}(T_{i,k}))$.

This means that $\theta_{i,k}(c) = \beta_{i_c,k}(c)$.

If $k_c = k - 1 \wedge i_c \geq i$: Substitution $\beta_{i_c,k-1}$ in $\beta_{i,k-1} \dots \beta_{n,k-1}$ changes c to $\beta_{i_c,k-1}(c)$. The substitution $\theta_{i,k-1}$ has no effect on $\beta_{i_c,k-1}(c)$ because members of $\mathsf{dom}(\theta_{i,k-1})$ are variables in D_{k-1} and can not be equal to $\beta_{i_c,k-1}(c)$.

$\mu_{m,k} \dots \mu_{g,k}$ also has no effect on $\beta_{i_c,k-1}(c)$ because we know that $c \in \mathsf{dom}(\mathsf{mgu}(T_{i,k}))$ and if $\beta_{i_c,k-1}(c)$ is a member of one of the domains in $\mu_{m,k} \dots \mu_{g,k}$ it contradicts Def. 5.1.2.iii).

$\beta_{i_c,k-1}(c)$ can not be in domain of any substitution in $\mu_{1,k+1} \dots \mu_{l,k+1}$, because from the second condition of our inductive step c should be in $\mu_{1,k} \dots \mu_{l,k}$ which contradict the third condition of Def. 5.1.2.iii).

This means that $\theta_{i,k}(c) = \beta_{i_c,k-1}(c)$.

So the substitution $\theta_{i,k}$ changes every member of $\mathsf{dom}(\mathsf{mgu}(T_{i,k}))$ to their next instances.

$\mathsf{ran}(\theta_{i,k}\mathsf{mgu}(T_{i,k}))$ also can not have any members of $\mathsf{dom}(\theta_{i,k}\mathsf{mgu}(T_{i,k}))$ because otherwise $\mathsf{mgu}(T_{i,k})$ contradicts Lemma 3.3.2. This means $\theta_{i,k}\mathsf{mgu}(T_{i,k}) = \mathsf{mgu}(\theta_{i,k}\mathsf{mgu}(T_{i,k}))$ and so $\mathsf{mgu}(\theta_{i,k}\mathsf{mgu}(T_{i,k}))$ is defined.

This proves that $R_{i,k+1}$ is applicable to S'_i also if R_i is primitive. As we saw, the members of $\mathsf{dom}(\mathsf{mgu}(\theta_{i,k}T_{i,k}))$ are the next instances of $\mathsf{dom}(\mathsf{mgu}(T_{i,k}))$, so the second condition of our inductive step still holds after the application of $R_{i,k+1}$ and the inductive step proof is done.

For the initial step we only need to show that our inductive step conditions are true for $i = 1$. From the theorem's condition we know that $\theta_{1,k}S_1 = S'_1$ so the first inductive condition is true. The second inductive condition is trivial because we do not have any primitive rules before R_1.

We proved that D_{k+1} can exist and we know that every domain member of $\mu_{j,k+1}$ in D_{k+1} is the next instance of a domain member in $\mu_{j,k}$. From this it is trivial that the second and third conditions of repetition candidates hold for D_{k+1}. The first condition is also trivial from our inductive step. So D_{k+1} is a repetition candidate. □

Corollary 5.1.4. *If in chain N_k derivation D_k is a repetition candidate we can continue the chain infinitely.*

Proof: From Theorem 5.1.3 we know that if D_k is derivation candidate, D_{k+1} can also exist and is a a repetition candidate. The same argument can be repeated this time for D_{k+1} and so on. So we can build infinite number of derivations for chain N_k.

The next lemma shows that after a finite number derivations in a chain, all the domain members of primitive rule applications are free variable instances and so we can check repetition candidate conditions.

Lemma 5.1.5. *In every infinite chain, after a finite number of derivations, for every primitive rule application $R_{i,k}$ all the members of* $\mathsf{dom}(\mathsf{mgu}(T_{i,k}))$ *are free variable instances.*

Proof: Every time a primitive rule is applied, a variable is replaced by a term in the constraint set. As the number of bound variables are finite, we can not have infinite number of primitive constraint applications with the bound variables in their domain. So from some point in the chain, bound variables as domain of primitive rule applications will never occur. □

Until now we showed that the occurrence of a certain type of derivation can cause infinite chains in constraint solver. On the other hand we already saw in previous chapter that a CHR system with only propagation rules is confluent. This means if a derivation candidate exists for a CHR system all the other derivations are also infinite.

Possible Completeness Argument

Based on the findings of this section we can build an algorithm that can find all the possible repetition candidates for a set of CHRs. But it does not mean that all non-terminating cases stem from repetition candidates. The completeness proof of our method is still an open problem and can be dealt with in two different levels.

We propose repetition of repetition candidates via Theorem 5.1.3 as the basic tool for detecting non-termination. We conjecture that it will be possible to weaken the premises of Theorem 5.1.3 by eliminating the repetition candidate conditions Def. 5.1.2.ii) and (iii).

Having the current or any other definition of repetition candidates with weaker conditions, the first step of completeness proof is to show that if for a derivation in a chain with all the primitive rule applications, $T_{i,k}$'s, having free variable instances as domain, the first condition of repetition candidate holds but other conditions are violated, then the chain can not be continued infinitely. The intuition behind the current conditions of repetition candidates is that if the domain variables of $T_{i,k}$ are not the last instances of free variables, they will not be replaced by their next instance after the application of $\beta_{i,k-1} \ldots \beta_{n,k-1} \cdot \beta_{1,k} \ldots \beta_{i-1,k}$ in $\theta_{i,k}$. This can be a potential danger as they can next be replaced by another term after the application of the second part of $\theta_{i,k}$ and make $\mathsf{mgu}(\theta_{i,k}T_{i,k})$ have no answers.

The second step of the completeness proof is to show that all non-terminating derivations are caused by the repetition of a finite derivation. This part of the proof can benefit from finiteness properties of CHR systems, such as finite number of rules, finite number of function symbols (type constructors in our case) and finite number of rule pairs that can be applied consecutively.

Even though we do not yet have the proofs for completeness, as we will see in the next section, the algorithm based on repetition candidate properties can detect non-termination for the examples presented in [SDPJS07] as the classic non-termination problems of violating FD conditions.

5.2 Finding Repetition Candidates

In previous sections we showed how repetition candidates can cause infinite derivations in the constraint solver. In this section we explain how from a set of CHRs all the possible repetition candidates can be found. Using this method we can find the constraint sets that can produce repetition candidates and detect these sets in the constraint solver.

The idea is to build a deduction tree from the rule set and explore different possible derivations among which all the derivations having repetition candidates as postfix exist as leaf nodes.

Definition 5.2.1. *A* deduction tree *for a set of CHRs is a tree with the following properties:*

i) Every node label is a derivation, $D: S_0 \xrightarrow{R_1} S_1 \ldots \xrightarrow{R_n} S_n$ *with the (omitted) derivation number equal to 1 where* $R_i: H_i \Rightarrow T_i$ $(0 \leq i \leq n)$. *The root node label is a derivation with an empty constraint set and no rules:* $D: S_0$, *where* $S_0 = \{\}$. *Also, if a rule is applied more than once in a derivation, rule instances used in the application are differentiated by variable renaming.*

ii) Every node with the label containing $D: S_0 \xrightarrow{R_1} S_1 \ldots \xrightarrow{R_n} S_n$, *has all the possible child nodes having the derivation with the format:*

$$D': (\alpha_0 \cdot \theta)(S \cup S_0) \xrightarrow{R_1} (\alpha_1 \cdot \theta)(S \cup S_1) \ldots \xrightarrow{R_n} (\alpha_n \cdot \theta)(S \cup S_n) \xrightarrow{R_{n+1}} S_{n+1}$$

which should satisfy the following conditions:

(a) $S \in H_{n+1}$ *where* $\exists \alpha.\ \alpha H_{n+1} \subseteq \alpha(S \cup S_n)$ *and* $\forall P.\ P \subset S \Rightarrow \nexists \alpha'.\ \alpha' H_{n+1} \subseteq \alpha'(P \cup S_n)$

(b) $Q \subseteq S_n \cup S$ *where* $\exists C \in Q. C \notin S_{n-1}$.

(c) $\theta = \gamma \cdot \beta$ *where:*

i. β *is defined as:* $\beta(Q) = \beta(H_{n+1}) \wedge \forall \beta'.\ (\beta'(Q) = \beta'(H_{n+1})) \Rightarrow (\exists \rho. \beta' = \rho \cdot \beta)$

ii. If R_{n+1} *is non-primitive* $\gamma = \{\}$.

iii. *If R_{n+1} is primitive γ is any substitution satisfying:*
$\gamma \subseteq p \wedge \mathsf{dom}(\gamma) = P_1 \wedge \mathsf{card}(\mathsf{dom}(\gamma)) = \mathsf{card}(P_2)$
where $p = P_1 \times P_2$ and $P_1 = \mathsf{Var}(S_1 \cap S)$ and P_2 is the set of all the bound variables in $\mathsf{Var}(\beta Q)$.

(d) *$\alpha_0 = \{\}$ and $\alpha_k = \alpha_{k-1}$ if R_k is non-primitive and $\alpha_k = \mathsf{mgu}((\alpha_{k-1} \cdot \theta)\{\!\{\mathsf{mgu}(T_{1,k})\}\!\}) \cdot \alpha_{k-1}$ if R_k is primitive.*

(e) *For every R_i $(1 \leq i \leq n)$, that is the same rule as R_{n+1}, there should be at least one primitive rule R_j $(i \leq j \leq n+1)$ and also for the constraint set $Q' \subseteq S_{i-1}$ underlying the the application of R_i we should have $\exists\theta.\ \theta(Q') = Q$.*

iii) *A node, N in the tree is a leaf node iff there exists a repetition candidate as the postfix of the derivation in the label of N or N can have no children.*

Def. 5.2.1.i) defines the label of nodes. Each node in a reduction tree has a derivation as the label and the root node's label is a derivation with only an empty constraint set. Def. 5.2.1.ii) shows the relation between a parent derivation and its child derivations. The idea is to make the "minimum" changes to the parent derivation to make its extension by another derivation step possible.

Two types of changes are made to the parent derivation for this purpose. As you can see in Def. 5.2.1.ii), first a constraint set S is added to every constraint set in the parent derivation. S is in fact the minimum necessary constraint set that should be added to the last constraint set of the parent derivation to make the next rule application possible. Notice that according to Def. 5.2.1.iia), we only add constraints when it is not possible to match the constraints in the rule head with unification. For the root derivation this is obviously the case, because no unification can match a rule head to an empty set.

The second change to the parent derivation is to apply the most general substitution that makes H_{n+1} and a subset of $S_n \cup S$, i.e. Q, equal. This substitution is defined in Def. 5.2.1.ii(c)i). Notice that this is different from matching the head to the constraint set as we do not apply the substitution

only to the constraint head. We are allowed to do that because we also have the option to change the constraint set to make it equal to the rule head and this in fact is how the second change to the parent derivation works.

The idea is to update the whole derivation in a way that at the end the last constraint set has the θ applied to it. To do this we apply θ to the first constraint set and by using the properties proved in Chapter 3 we show that this substitution is propagated through the derivation up to the last constraint set. We deal with this changing of substitutions by introducing α_is. Def. 5.2.1.iid) shows a recursive definition for the α_i substitutions applied in the child derivations. This formula can be obtained as follows.

Assume that we have the derivation step $S_1 \stackrel{R_k}{\rightarrow} S_2$, in which R is primitive and the derivation step $\alpha_{k-1} \cdot \theta S_1 \stackrel{R_k}{\rightarrow} \alpha_k \cdot \theta S_2$ exists. We want to find the relation between α_k and α_{k-1}. From the first derivation we know that for some substitution β we have $S_2 = \mathsf{mgu}(\beta T)S_1$. For the second derivation, the right constraint set would be equal to $\mathsf{mgu}(\alpha_{k-1} \cdot \theta \cdot \beta T) \cdot \alpha_{k-1} \cdot \theta S_1$. From Theorem 3.3.7, this is equal to:

$$\mathsf{mgu}(\alpha_{k-1} \cdot \theta \{\!\!\{ \mathsf{mgu}(\beta T) \}\!\!\}) \cdot \alpha_{k-1} \cdot \theta S_1$$

and from Corollary 3.3.4 it is equal to:

$$\mathsf{mgu}(\alpha_{k-1} \cdot \theta \{\!\!\{ \mathsf{mgu}(\beta T) \}\!\!\}) \cdot \alpha_{k-1} \cdot \theta \cdot \mathsf{mgu}(\beta T) S_1 = \mathsf{mgu}(\alpha_{k-1} \cdot \theta \{\!\!\{ \mathsf{mgu}(\beta T) \}\!\!\}) \cdot \alpha_{k-1} \cdot \theta S_2.$$

So we have $\alpha_k = \mathsf{mgu}(\alpha_{k-1} \cdot \theta \{\!\!\{ \mathsf{mgu}(\beta T) \}\!\!\}) \cdot \alpha_{k-1}$.

The second part of the θ substitution, γ, is only for the primitive rule applications and it deals with all the possible ways that variables in the first constraint set can be equal to the last constraint. This is important because we have to consider all the possible effects that the application of a primitive rule can have on a constraint set. Here we consider only the simple case of equality between variables. The more general dependencies between variables necessitates introducing second order variables and higher order unification which is not discussed here due to its complications.

Updating the derivation to make the new derivation step possible sometimes results infinite branches even though the rule set can not cause infinite derivations itself. As a simple example consider a rule set with a single rule

$R : C\ [x] \Rightarrow C\ x$. This rule can never cause infinite derivations. but when we are making a deduction tree, at every step a substitution θ can update the derivation and make the next application of R possible and this can go on infinitely. Def. 5.2.1.iie) avoids these situations by restricting application of a rule more that once in each derivation. The only case that allows more that one application of the same rule is when between the two rules there exists a primitive rule and the first constraint set of derivation between the the two rules matches with the last constraint set. In this case, the repetition of the same rule sequence can result a derivation candidate, and we should allow the rule application, otherwise we might miss a repetition candidate that can be produced by the rule set.

Finally Def. 5.2.1.iii) defines leaves of a reduction tree. Every node in the tree with a derivation that is not extendable is a node. Also if the derivation of a node has a repetition candidate as postfix it is also a leaf. Derivations with repetition candidates as postfix are in fact what we were looking for. Having them, we can avoid infinite chains by finding the constraint sets that can lead to repetition candidates (these constraint sets are in facts the first constraint sets of derivations having repetition candidates).

5.2.1 Deduction Trees Finiteness

To be of practical use deduction trees should be finite, otherwise any algorithm building them would be non-terminating. The first part of the finiteness argument for deduction trees is based on the completeness argument that we had for repetition candidate. Assuming that repetition candidates are the only reason that infinite derivations happen, we can not have a deduction tree with an infinite branch. Because if the branch is infinite, it should have a repetition candidate inside it and we have a checking in our definition of deduction trees to stop extending the derivation more when we reach a repetition candidate. Also as we explained before, Def. 5.2.1.iie) prevents updating derivations and adding new derivation steps infinitely.

The number of branches for every node is also always finite. Because what determines the number of branches is the number of rules in the system and also the number of γ substitutions which are both finite.

5.2.2 Building the Deduction Tree

The general strategy to build a deduction tree from a set of rules is to start from the root with an empty constraint set as its derivation and perform a BFS or DFS traversal. At each step a new child is added by updating the parent derivation and extending it with a new derivation step. To update the parent derivation, a new rule is first picked to be applied to the last constraint set. Next S in Def. 5.2.1.ii) is found using Def. 5.2.1.iia). After having S, substitution β is found by using Def. 5.2.1.ii(c)i). If the rule is primitive γ substitutions satisfying Def. 5.2.1.ii(c)iii) and Def. 5.2.1.ii(c)ii) are next found. As there can exist more than one γ substitution, we can have more than one child by the same rule application. After finding γ we use the recursive definition for α_i to find all the $\alpha_0 \dots \alpha_n$ substitutions.

Also, every algorithm that builds a deduction tree needs to use unification and matching algorithms. There are standard algorithms in term rewriting systems literature that we can use for unification and matching [Klo92], but the only problem is how we should deal with free variables. The point is that even though free variables are a type of variable in CHR systems, but when unifying two constraints they should not be substituted with a term so we treat free variables as constants (type constructors with no parameters) when applying unifying and matching rules.

5.2.3 Example

In this section we build the deduction tree for the example presented in [SDPJS07] as the case that not obeying FD conditions causes non-termination. From the class and instance declarations and the generated CHRs presented next a deduction tree can be generated.

```
class D a
class F a b | a -> b
instance F [a] [[a]]
instance (D c, F a c) => D [a]
```

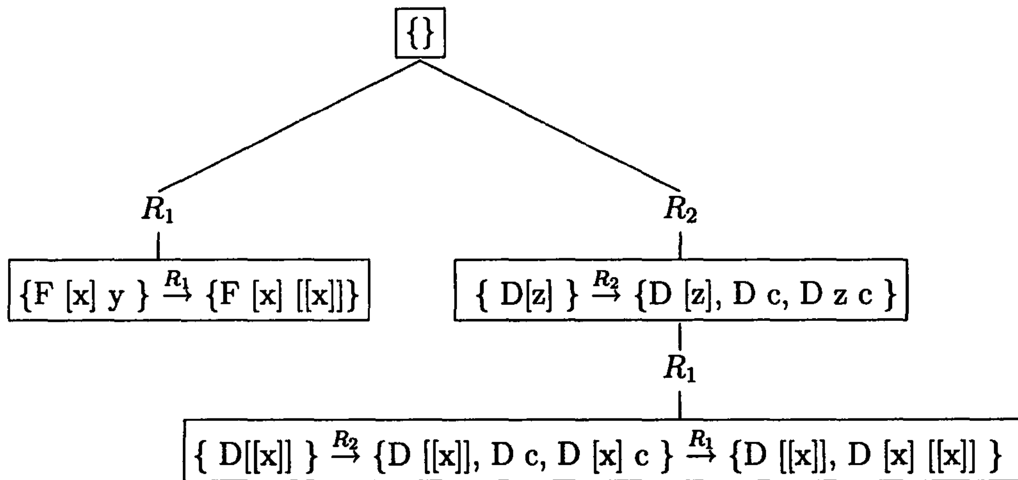

Figure 5.1: Deduction Tree

R_1: F [x] y $\Rightarrow$ y =[[x]]
R_2: D [z] $\Rightarrow$ D c, F z c

We start with an empty set and apply R_1 and R_2 to it. The left branch can not be extended any more because there is no other rule to be applied to the last constraint set of the derivation and for the right branch we have a repetition candidate as the leaf node. This means that for this set of rules if we start with any instance of the constraint set $\{D\ [[x]]\}$, we will have a non-terminating derivation.

Chapter 6

Conclusions

CHRs have proven to be a useful tool to formalize functional dependencies in type systems. But restrictions on FD definitions in order to make the resulting CHR system terminating and confluent greatly affect the benefits of this approach.

As an alternative way to deal with the problem of confluence, we introduced prioritized CHRs and showed how applying propagation rules prior to simplification rules can solve the problem of confluence in CHRs without affecting the semantics.

Based on this modified system of rule applications, we studied the properties of non-terminating CHR systems. We built a formal framework that included all the main characteristics of CHRs. Based upon that, we defined a special type of derivation named repetition candidate and proved that if a repetition candidate exists, we can build an infinite derivation by repeating application of the same rule sequence in the repetition candidate. Based on this property and the confluence of propagation rules discussed in Chapter 4, we showed that the existence of a repetition candidate makes a CHR system non-terminating.

Finally, we introduced *deduction trees* as a way to find all the possible repetition candidates for a set of CHRs. This algorithm can be applied to the CHRs generated from the type class and instance declarations and enables us to judge whether a class or instance declaration causes non-termination.

There are in fact two distinguishable types of repetition candidates in the algorithm results. If the first constraint set of the derivation that has a repetition candidate exactly matches head constraints of the first rule (except for variable renaming), it means any application of this rule will cause non-termination. In this case, the instance or class declaration which produced the rules in the repetition candidate should be rejected.

But in many cases, such as the examples we presented in the previous chapter, the first constraint set is an instance of the rule head but not exactly the same. In this case we have two options, either reject the class or instance declaration, or postpone the error until instances of any constraint set in the repetition candidate are generated during rule applications. In this way more class definitions are accepted as valid and we can only prevent wrong usages of type class member functions in the code body.

6.1 Contributions

- By finding the initial constraint sets that can cause non-termination in CHRs generated from class and instance definitions, we can remove the FD conditions and reject only those function definitions that can produce those initial constraints. This gives us a more expressive type system by only checking some extra conditions in the constraint solver, without the need to change the current type inference system.

- As, in general, detecting non-termination in term rewriting systems is not decidable, the main focus has always been on finding sufficient conditions to ensure a term rewriting system is terminating. But, by focusing on a special type of term rewriting system we showed that working on non-terminating systems can result in interesting observations and help us to understand how exactly non-termination might happen.

- The findings of this thesis are not limited to the application of CHRs in formalizing type dependencies. Any other system that works with CHR systems with only equality as the primitive constraint symbol can use the presented algorithm to detect cases that cause non-termination in

constraint solvers.

- Having free variables is usually rejected in term rewriting systems because of their unwanted consequences. In this research we presented a formal definition for these types of variables that enabled us to study their behavior. This formalization can also have applications in other term rewriting systems.
- Substitutions have never been interpreted as primitive constraint sets, as far as we know. This new approach can also be used in other term rewriting systems to discover new properties for these systems.
- The formal framework we built in Chapter 3 can also be used to study other properties of CHRs.
- Some of the strategies and theorems used to formalize CHRs are general enough to be useful for other constraint solving systems.

6.2 Future Work

In Chapter 5 we showed how a sequence of rule applications can repeat infinitely and cause non-termination. But proving that all the non-termination cases stem from a repetition candidate is still an open problem. We believe that our formulation of the problem can be a good starting point to work on the completeness proof. Furthermore, even without the completeness proof, our algorithm can still be useful because practically it covers the known non-termination cases.

Also, in the deduction tree definition discussed in Chapter 5, we only considered the simple case of equality between different variables. But this does not cover all the cases that might happen. In general if we have two sets of variables $S = \{x_1 \dots x_n\}$ and $S' = \{y_1 \dots y_m\}$ (S is the first constraint set and S' is the set of all the bound variables in the primitive rule application), we will have the substitution $\alpha = \{x_1 \mapsto v_1(y_1 \dots y_m) \dots x_n \mapsto v_n(y_1 \dots y_m)\}$ in which $v_1 \dots v_n$ are second order variables, and so the unification algorithm used to update the parent node derivation should support second order unification.

Bibliography

[Abd97] Slim Abdennadher. Operational semantics and confluence of constraint propagation rules. *In Proc. of the Third International Conference on Principles and Practice of Constraint Programming*, 1997.

[CKPJ05] Manuel M. T. Chakravarty, Gabriele Keller, and Simon L. Peyton Jones. Associated type synonyms. *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 241–253, 2005.

[CKPJM05] Manuel M. T. Chakravarty, Gabriele Keller, Simon L. Peyton Jones, and Simon Marlow. Associated types with class. *ACM Conference on Principles of Programming Languages*, pages 1–13, 2005.

[Frü98] Thom Frühwirth. Theory and practice of constraint handling rules. *Special issue on constraint logic programming. Journal of Logic Programming*, 37(1–3):95–138, 1998.

[HHPJW96] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in haskell. *ACM Transactions on Programming Languages*, 18:109–138, 1996.

[Jon93] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *FPCA '93: Conference on Functional Programming and Computer Architecture, Copenhagen, Denmark*, pages 52–61, New York, N.Y., 1993. ACM Press.

[Jon00] Mark P. Jones. Type classes with functional dependencies. In G. Smolka, editor, *ESOP 2000*, volume 1782 of *LNCS*, pages 230–244. Springer, March 2000.

[Klo92] J.W. Klop. Term rewriting systems. *Handbook of Logic in Computer Science*, 2:1–116, 1992.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

[PJJM97] Simon L. Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. In *Proc. Haskell Workshop 1997*, 1997.

[SDPJS07] Martin Sulzmann, Gregory J. Duck, Simon L. Peyton Jones, and Peter J. Stuckey. Understading functional dependencies via constraint handling rules. *Journal of Functional Programming*, 17:83–129, 2007.

[SS02] Peter J. Stuckey and Martin Sulzmann. A theory of overloading. *In Proc. of ICFP'02*, pages 167–178, 2002.

[Ter03] Terese, editor. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts Theoret. Comput. Sci.* Cambridge Univ. Press, 2003.

[Wik] Wikipedia. Constraint programming.
http://en.wikipedia.org/wiki/Constraint_programming.

3071 10